Trusted Platform Module Library
Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

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 by 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 TPM configuration is largely defined by TpmProfiles.h. This file may be edited in order to change the algorithms and commands supported by a TPM implementation.

A parser exists to process a Word document that defines the TPM configuration. This parser is used to create TpmProfiles.h.

4.2 Structure Parser

4.2.1 Introduction

The program that processes the tables in TPM 2.0 Part 2 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);
```

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

The most recent versions of the TPM code includes the option to use table-driven marshaling rather that the procedural marshaling described in previous clauses in 4.2.2. The structure and processing of this code is complex and is provided in the code.

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   #if CC_Create // Command must be enabled
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)
```

```
#define RC_Create_inSensitive   (TPM_RC_P + TPM_RC_1)
#define RC_Create_inPublic      (TPM_RC_P + TPM_RC_2)
#define RC_Create_outsideInfo   (TPM_RC_P + TPM_RC_3)
#define RC_Create_creationPCR   (TPM_RC_P + TPM_RC_4)
```

Function prototype

```
TPM_RC
TPM2_Create(
    Create_In                   *in,
    Create_Out                  *out
);
#endif  // _Create_FP_H_
#endif  // CC_Create
```

4.5 Portability

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

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

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

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

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

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

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

5 Header Files

5.1 Introduction

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

5.2 BaseTypes.h

```
1   #ifndef _BASE_TYPES_H_
2   #define _BASE_TYPES_H_
```

NULL definition

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

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


5.4 CommandAttributeData.h

This file should only be included by CommandCodeAttibutes.c

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

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

```
const TPMA_CC    s_ccAttr [] = {
#if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
        TPMA_CC_INITIALIZER(0x011F, 0, 1, 0, 0, 2, 0, 0, 0),
#endif
#if (PAD_LIST || CC_EvictControl)
        TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 0, 2, 0, 0, 0),
#endif
#if (PAD_LIST || CC_HierarchyControl)
        TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_NV_UndefineSpace)
        TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 0, 2, 0, 0, 0),
#endif
#if (PAD_LIST )
        TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0, 0),
#endif
#if (PAD_LIST || CC_ChangeEPS)
        TPMA_CC_INITIALIZER(0x0124, 0, 1, 1, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_ChangePPS)
        TPMA_CC_INITIALIZER(0x0125, 0, 1, 1, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_Clear)
        TPMA_CC_INITIALIZER(0x0126, 0, 1, 1, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_ClearControl)
        TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_ClockSet)
        TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_HierarchyChangeAuth)
        TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_NV_DefineSpace)
        TPMA_CC_INITIALIZER(0x012A, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_PCR_Allocate)
        TPMA_CC_INITIALIZER(0x012B, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_PCR_SetAuthPolicy)
        TPMA_CC_INITIALIZER(0x012C, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_PP_Commands)
        TPMA_CC_INITIALIZER(0x012D, 0, 1, 0, 0, 1, 0, 0, 0),
#endif
#if (PAD_LIST || CC_SetPrimaryPolicy)
        TPMA_CC_INITIALIZER(0x012E, 0, 1, 0, 0, 1, 0, 0, 0),
#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_CertifyX509)
370           TPMA_CC_INITIALIZER(0x0197, 0, 0, 0, 0, 2, 0, 0, 0),
371   #endif
372   #if (PAD_LIST || CC_ACT_SetTimeout)
373           TPMA_CC_INITIALIZER(0x0198, 0, 0, 0, 0, 1, 0, 0, 0),
374   #endif
375   #if (PAD_LIST || CC_Vendor_TCG_Test)
376           TPMA_CC_INITIALIZER(0x0000, 0, 0, 0, 0, 0, 0, 1, 0),
377   #endif
378           TPMA_ZERO_INITIALIZER()
379   };
```

This is the command code attribute structure.

```
380   const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
381   #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
382           (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial     *  // 0x011F
```

```
383             (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
384 #endif
385 #if (PAD_LIST || CC_EvictControl)
386         (COMMAND_ATTRIBUTES)(CC_EvictControl                *  // 0x0120
387             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
388 #endif
389 #if (PAD_LIST || CC_HierarchyControl)
390         (COMMAND_ATTRIBUTES)(CC_HierarchyControl            *  // 0x0121
391             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
392 #endif
393 #if (PAD_LIST || CC_NV_UndefineSpace)
394         (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace            *  // 0x0122
395             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
396 #endif
397 #if (PAD_LIST )
398         (COMMAND_ATTRIBUTES)(0),                               // 0x0123
399 #endif
400 #if (PAD_LIST || CC_ChangeEPS)
401         (COMMAND_ATTRIBUTES)(CC_ChangeEPS                   *  // 0x0124
402             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
403 #endif
404 #if (PAD_LIST || CC_ChangePPS)
405         (COMMAND_ATTRIBUTES)(CC_ChangePPS                   *  // 0x0125
406             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
407 #endif
408 #if (PAD_LIST || CC_Clear)
409         (COMMAND_ATTRIBUTES)(CC_Clear                       *  // 0x0126
410             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
411 #endif
412 #if (PAD_LIST || CC_ClearControl)
413         (COMMAND_ATTRIBUTES)(CC_ClearControl                *  // 0x0127
414             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
415 #endif
416 #if (PAD_LIST || CC_ClockSet)
417         (COMMAND_ATTRIBUTES)(CC_ClockSet                    *  // 0x0128
418             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
419 #endif
420 #if (PAD_LIST || CC_HierarchyChangeAuth)
421         (COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth         *  // 0x0129
422             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
423 #endif
424 #if (PAD_LIST || CC_NV_DefineSpace)
425         (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace              *  // 0x012A
426             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
427 #endif
428 #if (PAD_LIST || CC_PCR_Allocate)
429         (COMMAND_ATTRIBUTES)(CC_PCR_Allocate                *  // 0x012B
430             (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
431 #endif
432 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
433         (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthPolicy           *  // 0x012C
434             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
435 #endif
436 #if (PAD_LIST || CC_PP_Commands)
437         (COMMAND_ATTRIBUTES)(CC_PP_Commands                 *  // 0x012D
438             (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
439 #endif
440 #if (PAD_LIST || CC_SetPrimaryPolicy)
441         (COMMAND_ATTRIBUTES)(CC_SetPrimaryPolicy            *  // 0x012E
442             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
443 #endif
444 #if (PAD_LIST || CC_FieldUpgradeStart)
445         (COMMAND_ATTRIBUTES)(CC_FieldUpgradeStart           *  // 0x012F
446             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
447 #endif
448 #if (PAD_LIST || CC_ClockRateAdjust)
```

```
449             (COMMAND_ATTRIBUTES)(CC_ClockRateAdjust             *  // 0x0130
450                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
451     #endif
452     #if (PAD_LIST || CC_CreatePrimary)
453             (COMMAND_ATTRIBUTES)(CC_CreatePrimary               *  // 0x0131
454                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
455     #endif
456     #if (PAD_LIST || CC_NV_GlobalWriteLock)
457             (COMMAND_ATTRIBUTES)(CC_NV_GlobalWriteLock          *  // 0x0132
458                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
459     #endif
460     #if (PAD_LIST || CC_GetCommandAuditDigest)
461             (COMMAND_ATTRIBUTES)(CC_GetCommandAuditDigest       *  // 0x0133
462                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
463     #endif
464     #if (PAD_LIST || CC_NV_Increment)
465             (COMMAND_ATTRIBUTES)(CC_NV_Increment                *  // 0x0134
466                 (IS_IMPLEMENTED+HANDLE_1_USER)),
467     #endif
468     #if (PAD_LIST || CC_NV_SetBits)
469             (COMMAND_ATTRIBUTES)(CC_NV_SetBits                  *  // 0x0135
470                 (IS_IMPLEMENTED+HANDLE_1_USER)),
471     #endif
472     #if (PAD_LIST || CC_NV_Extend)
473             (COMMAND_ATTRIBUTES)(CC_NV_Extend                   *  // 0x0136
474                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
475     #endif
476     #if (PAD_LIST || CC_NV_Write)
477             (COMMAND_ATTRIBUTES)(CC_NV_Write                    *  // 0x0137
478                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
479     #endif
480     #if (PAD_LIST || CC_NV_WriteLock)
481             (COMMAND_ATTRIBUTES)(CC_NV_WriteLock                *  // 0x0138
482                 (IS_IMPLEMENTED+HANDLE_1_USER)),
483     #endif
484     #if (PAD_LIST || CC_DictionaryAttackLockReset)
485             (COMMAND_ATTRIBUTES)(CC_DictionaryAttackLockReset   *  // 0x0139
486                 (IS_IMPLEMENTED+HANDLE_1_USER)),
487     #endif
488     #if (PAD_LIST || CC_DictionaryAttackParameters)
489             (COMMAND_ATTRIBUTES)(CC_DictionaryAttackParameters  *  // 0x013A
490                 (IS_IMPLEMENTED+HANDLE_1_USER)),
491     #endif
492     #if (PAD_LIST || CC_NV_ChangeAuth)
493             (COMMAND_ATTRIBUTES)(CC_NV_ChangeAuth               *  // 0x013B
494                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
495     #endif
496     #if (PAD_LIST || CC_PCR_Event)
497             (COMMAND_ATTRIBUTES)(CC_PCR_Event                   *  // 0x013C
498                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
499     #endif
500     #if (PAD_LIST || CC_PCR_Reset)
501             (COMMAND_ATTRIBUTES)(CC_PCR_Reset                   *  // 0x013D
502                 (IS_IMPLEMENTED+HANDLE_1_USER)),
503     #endif
504     #if (PAD_LIST || CC_SequenceComplete)
505             (COMMAND_ATTRIBUTES)(CC_SequenceComplete            *  // 0x013E
506                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
507     #endif
508     #if (PAD_LIST || CC_SetAlgorithmSet)
509             (COMMAND_ATTRIBUTES)(CC_SetAlgorithmSet             *  // 0x013F
510                 (IS_IMPLEMENTED+HANDLE_1_USER)),
511     #endif
512     #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
513             (COMMAND_ATTRIBUTES)(CC_SetCommandCodeAuditStatus   *  // 0x0140
514                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
```

```
515   #endif
516   #if (PAD_LIST || CC_FieldUpgradeData)
517           (COMMAND_ATTRIBUTES)(CC_FieldUpgradeData            *  // 0x0141
518               (IS_IMPLEMENTED+DECRYPT_2)),
519   #endif
520   #if (PAD_LIST || CC_IncrementalSelfTest)
521           (COMMAND_ATTRIBUTES)(CC_IncrementalSelfTest         *  // 0x0142
522               (IS_IMPLEMENTED)),
523   #endif
524   #if (PAD_LIST || CC_SelfTest)
525           (COMMAND_ATTRIBUTES)(CC_SelfTest                    *  // 0x0143
526               (IS_IMPLEMENTED)),
527   #endif
528   #if (PAD_LIST || CC_Startup)
529           (COMMAND_ATTRIBUTES)(CC_Startup                     *  // 0x0144
530               (IS_IMPLEMENTED+NO_SESSIONS)),
531   #endif
532   #if (PAD_LIST || CC_Shutdown)
533           (COMMAND_ATTRIBUTES)(CC_Shutdown                    *  // 0x0145
534               (IS_IMPLEMENTED)),
535   #endif
536   #if (PAD_LIST || CC_StirRandom)
537           (COMMAND_ATTRIBUTES)(CC_StirRandom                  *  // 0x0146
538               (IS_IMPLEMENTED+DECRYPT_2)),
539   #endif
540   #if (PAD_LIST || CC_ActivateCredential)
541           (COMMAND_ATTRIBUTES)(CC_ActivateCredential          *  // 0x0147
542               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
543   #endif
544   #if (PAD_LIST || CC_Certify)
545           (COMMAND_ATTRIBUTES)(CC_Certify                     *  // 0x0148
546               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
547   #endif
548   #if (PAD_LIST || CC_PolicyNV)
549           (COMMAND_ATTRIBUTES)(CC_PolicyNV                    *  // 0x0149
550               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
551   #endif
552   #if (PAD_LIST || CC_CertifyCreation)
553           (COMMAND_ATTRIBUTES)(CC_CertifyCreation             *  // 0x014A
554               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
555   #endif
556   #if (PAD_LIST || CC_Duplicate)
557           (COMMAND_ATTRIBUTES)(CC_Duplicate                   *  // 0x014B
558               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
559   #endif
560   #if (PAD_LIST || CC_GetTime)
561           (COMMAND_ATTRIBUTES)(CC_GetTime                     *  // 0x014C
562               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
563   #endif
564   #if (PAD_LIST || CC_GetSessionAuditDigest)
565           (COMMAND_ATTRIBUTES)(CC_GetSessionAuditDigest       *  // 0x014D
566               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
567   #endif
568   #if (PAD_LIST || CC_NV_Read)
569           (COMMAND_ATTRIBUTES)(CC_NV_Read                     *  // 0x014E
570               (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
571   #endif
572   #if (PAD_LIST || CC_NV_ReadLock)
573           (COMMAND_ATTRIBUTES)(CC_NV_ReadLock                 *  // 0x014F
574               (IS_IMPLEMENTED+HANDLE_1_USER)),
575   #endif
576   #if (PAD_LIST || CC_ObjectChangeAuth)
577           (COMMAND_ATTRIBUTES)(CC_ObjectChangeAuth            *  // 0x0150
578               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
579   #endif
580   #if (PAD_LIST || CC_PolicySecret)
```

```
581             (COMMAND_ATTRIBUTES)(CC_PolicySecret                    *  // 0x0151
582                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
583     #endif
584     #if (PAD_LIST || CC_Rewrap)
585             (COMMAND_ATTRIBUTES)(CC_Rewrap                          *  // 0x0152
586                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
587     #endif
588     #if (PAD_LIST || CC_Create)
589             (COMMAND_ATTRIBUTES)(CC_Create                          *  // 0x0153
590                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
591     #endif
592     #if (PAD_LIST || CC_ECDH_ZGen)
593             (COMMAND_ATTRIBUTES)(CC_ECDH_ZGen                       *  // 0x0154
594                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
595     #endif
596     #if (PAD_LIST || (CC_HMAC || CC_MAC))
597             (COMMAND_ATTRIBUTES)((CC_HMAC || CC_MAC)                *  // 0x0155
598                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
599     #endif
600     #if (PAD_LIST || CC_Import)
601             (COMMAND_ATTRIBUTES)(CC_Import                          *  // 0x0156
602                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
603     #endif
604     #if (PAD_LIST || CC_Load)
605             (COMMAND_ATTRIBUTES)(CC_Load                            *  // 0x0157
606                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
607     #endif
608     #if (PAD_LIST || CC_Quote)
609             (COMMAND_ATTRIBUTES)(CC_Quote                           *  // 0x0158
610                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
611     #endif
612     #if (PAD_LIST || CC_RSA_Decrypt)
613             (COMMAND_ATTRIBUTES)(CC_RSA_Decrypt                     *  // 0x0159
614                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
615     #endif
616     #if (PAD_LIST )
617             (COMMAND_ATTRIBUTES)(0),                                   // 0x015A
618     #endif
619     #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
620             (COMMAND_ATTRIBUTES)((CC_HMAC_Start || CC_MAC_Start) *  // 0x015B
621                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
622     #endif
623     #if (PAD_LIST || CC_SequenceUpdate)
624             (COMMAND_ATTRIBUTES)(CC_SequenceUpdate                  *  // 0x015C
625                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
626     #endif
627     #if (PAD_LIST || CC_Sign)
628             (COMMAND_ATTRIBUTES)(CC_Sign                            *  // 0x015D
629                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
630     #endif
631     #if (PAD_LIST || CC_Unseal)
632             (COMMAND_ATTRIBUTES)(CC_Unseal                          *  // 0x015E
633                 (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
634     #endif
635     #if (PAD_LIST )
636             (COMMAND_ATTRIBUTES)(0),                                   // 0x015F
637     #endif
638     #if (PAD_LIST || CC_PolicySigned)
639             (COMMAND_ATTRIBUTES)(CC_PolicySigned                    *  // 0x0160
640                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
641     #endif
642     #if (PAD_LIST || CC_ContextLoad)
643             (COMMAND_ATTRIBUTES)(CC_ContextLoad                     *  // 0x0161
644                 (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
645     #endif
646     #if (PAD_LIST || CC_ContextSave)
```

```
647             (COMMAND_ATTRIBUTES)(CC_ContextSave                 *  // 0x0162
648                 (IS_IMPLEMENTED+NO_SESSIONS)),
649     #endif
650     #if (PAD_LIST || CC_ECDH_KeyGen)
651             (COMMAND_ATTRIBUTES)(CC_ECDH_KeyGen                 *  // 0x0163
652                 (IS_IMPLEMENTED+ENCRYPT_2)),
653     #endif
654     #if (PAD_LIST || CC_EncryptDecrypt)
655             (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt              *  // 0x0164
656                 (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
657     #endif
658     #if (PAD_LIST || CC_FlushContext)
659             (COMMAND_ATTRIBUTES)(CC_FlushContext                *  // 0x0165
660                 (IS_IMPLEMENTED+NO_SESSIONS)),
661     #endif
662     #if (PAD_LIST )
663             (COMMAND_ATTRIBUTES)(0),                               // 0x0166
664     #endif
665     #if (PAD_LIST || CC_LoadExternal)
666             (COMMAND_ATTRIBUTES)(CC_LoadExternal                *  // 0x0167
667                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
668     #endif
669     #if (PAD_LIST || CC_MakeCredential)
670             (COMMAND_ATTRIBUTES)(CC_MakeCredential              *  // 0x0168
671                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
672     #endif
673     #if (PAD_LIST || CC_NV_ReadPublic)
674             (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic               *  // 0x0169
675                 (IS_IMPLEMENTED+ENCRYPT_2)),
676     #endif
677     #if (PAD_LIST || CC_PolicyAuthorize)
678             (COMMAND_ATTRIBUTES)(CC_PolicyAuthorize             *  // 0x016A
679                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
680     #endif
681     #if (PAD_LIST || CC_PolicyAuthValue)
682             (COMMAND_ATTRIBUTES)(CC_PolicyAuthValue             *  // 0x016B
683                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
684     #endif
685     #if (PAD_LIST || CC_PolicyCommandCode)
686             (COMMAND_ATTRIBUTES)(CC_PolicyCommandCode           *  // 0x016C
687                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
688     #endif
689     #if (PAD_LIST || CC_PolicyCounterTimer)
690             (COMMAND_ATTRIBUTES)(CC_PolicyCounterTimer          *  // 0x016D
691                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
692     #endif
693     #if (PAD_LIST || CC_PolicyCpHash)
694             (COMMAND_ATTRIBUTES)(CC_PolicyCpHash                *  // 0x016E
695                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
696     #endif
697     #if (PAD_LIST || CC_PolicyLocality)
698             (COMMAND_ATTRIBUTES)(CC_PolicyLocality              *  // 0x016F
699                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
700     #endif
701     #if (PAD_LIST || CC_PolicyNameHash)
702             (COMMAND_ATTRIBUTES)(CC_PolicyNameHash              *  // 0x0170
703                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
704     #endif
705     #if (PAD_LIST || CC_PolicyOR)
706             (COMMAND_ATTRIBUTES)(CC_PolicyOR                    *  // 0x0171
707                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
708     #endif
709     #if (PAD_LIST || CC_PolicyTicket)
710             (COMMAND_ATTRIBUTES)(CC_PolicyTicket                *  // 0x0172
711                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
712     #endif
```

```
713   #if (PAD_LIST || CC_ReadPublic)
714           (COMMAND_ATTRIBUTES)(CC_ReadPublic                  *  // 0x0173
715               (IS_IMPLEMENTED+ENCRYPT_2)),
716   #endif
717   #if (PAD_LIST || CC_RSA_Encrypt)
718           (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt                 *  // 0x0174
719               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
720   #endif
721   #if (PAD_LIST )
722           (COMMAND_ATTRIBUTES)(0),                               // 0x0175
723   #endif
724   #if (PAD_LIST || CC_StartAuthSession)
725           (COMMAND_ATTRIBUTES)(CC_StartAuthSession            *  // 0x0176
726               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
727   #endif
728   #if (PAD_LIST || CC_VerifySignature)
729           (COMMAND_ATTRIBUTES)(CC_VerifySignature             *  // 0x0177
730               (IS_IMPLEMENTED+DECRYPT_2)),
731   #endif
732   #if (PAD_LIST || CC_ECC_Parameters)
733           (COMMAND_ATTRIBUTES)(CC_ECC_Parameters              *  // 0x0178
734               (IS_IMPLEMENTED)),
735   #endif
736   #if (PAD_LIST || CC_FirmwareRead)
737           (COMMAND_ATTRIBUTES)(CC_FirmwareRead                *  // 0x0179
738               (IS_IMPLEMENTED+ENCRYPT_2)),
739   #endif
740   #if (PAD_LIST || CC_GetCapability)
741           (COMMAND_ATTRIBUTES)(CC_GetCapability               *  // 0x017A
742               (IS_IMPLEMENTED)),
743   #endif
744   #if (PAD_LIST || CC_GetRandom)
745           (COMMAND_ATTRIBUTES)(CC_GetRandom                   *  // 0x017B
746               (IS_IMPLEMENTED+ENCRYPT_2)),
747   #endif
748   #if (PAD_LIST || CC_GetTestResult)
749           (COMMAND_ATTRIBUTES)(CC_GetTestResult               *  // 0x017C
750               (IS_IMPLEMENTED+ENCRYPT_2)),
751   #endif
752   #if (PAD_LIST || CC_Hash)
753           (COMMAND_ATTRIBUTES)(CC_Hash                        *  // 0x017D
754               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
755   #endif
756   #if (PAD_LIST || CC_PCR_Read)
757           (COMMAND_ATTRIBUTES)(CC_PCR_Read                    *  // 0x017E
758               (IS_IMPLEMENTED)),
759   #endif
760   #if (PAD_LIST || CC_PolicyPCR)
761           (COMMAND_ATTRIBUTES)(CC_PolicyPCR                   *  // 0x017F
762               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
763   #endif
764   #if (PAD_LIST || CC_PolicyRestart)
765           (COMMAND_ATTRIBUTES)(CC_PolicyRestart               *  // 0x0180
766               (IS_IMPLEMENTED+ALLOW_TRIAL)),
767   #endif
768   #if (PAD_LIST || CC_ReadClock)
769           (COMMAND_ATTRIBUTES)(CC_ReadClock                   *  // 0x0181
770               (IS_IMPLEMENTED)),
771   #endif
772   #if (PAD_LIST || CC_PCR_Extend)
773           (COMMAND_ATTRIBUTES)(CC_PCR_Extend                  *  // 0x0182
774               (IS_IMPLEMENTED+HANDLE_1_USER)),
775   #endif
776   #if (PAD_LIST || CC_PCR_SetAuthValue)
777           (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthValue            *  // 0x0183
778               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
```

```
779 #endif
780 #if (PAD_LIST || CC_NV_Certify)
781         (COMMAND_ATTRIBUTES)(CC_NV_Certify                  *  // 0x0184
782             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
783 #endif
784 #if (PAD_LIST || CC_EventSequenceComplete)
785         (COMMAND_ATTRIBUTES)(CC_EventSequenceComplete       *  // 0x0185
786             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
787 #endif
788 #if (PAD_LIST || CC_HashSequenceStart)
789         (COMMAND_ATTRIBUTES)(CC_HashSequenceStart           *  // 0x0186
790             (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
791 #endif
792 #if (PAD_LIST || CC_PolicyPhysicalPresence)
793         (COMMAND_ATTRIBUTES)(CC_PolicyPhysicalPresence      *  // 0x0187
794             (IS_IMPLEMENTED+ALLOW_TRIAL)),
795 #endif
796 #if (PAD_LIST || CC_PolicyDuplicationSelect)
797         (COMMAND_ATTRIBUTES)(CC_PolicyDuplicationSelect     *  // 0x0188
798             (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
799 #endif
800 #if (PAD_LIST || CC_PolicyGetDigest)
801         (COMMAND_ATTRIBUTES)(CC_PolicyGetDigest             *  // 0x0189
802             (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
803 #endif
804 #if (PAD_LIST || CC_TestParms)
805         (COMMAND_ATTRIBUTES)(CC_TestParms                   *  // 0x018A
806             (IS_IMPLEMENTED)),
807 #endif
808 #if (PAD_LIST || CC_Commit)
809         (COMMAND_ATTRIBUTES)(CC_Commit                      *  // 0x018B
810             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
811 #endif
812 #if (PAD_LIST || CC_PolicyPassword)
813         (COMMAND_ATTRIBUTES)(CC_PolicyPassword              *  // 0x018C
814             (IS_IMPLEMENTED+ALLOW_TRIAL)),
815 #endif
816 #if (PAD_LIST || CC_ZGen_2Phase)
817         (COMMAND_ATTRIBUTES)(CC_ZGen_2Phase                 *  // 0x018D
818             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
819 #endif
820 #if (PAD_LIST || CC_EC_Ephemeral)
821         (COMMAND_ATTRIBUTES)(CC_EC_Ephemeral                *  // 0x018E
822             (IS_IMPLEMENTED+ENCRYPT_2)),
823 #endif
824 #if (PAD_LIST || CC_PolicyNvWritten)
825         (COMMAND_ATTRIBUTES)(CC_PolicyNvWritten             *  // 0x018F
826             (IS_IMPLEMENTED+ALLOW_TRIAL)),
827 #endif
828 #if (PAD_LIST || CC_PolicyTemplate)
829         (COMMAND_ATTRIBUTES)(CC_PolicyTemplate              *  // 0x0190
830             (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
831 #endif
832 #if (PAD_LIST || CC_CreateLoaded)
833         (COMMAND_ATTRIBUTES)(CC_CreateLoaded                *  // 0x0191
834             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
835 #endif
836 #if (PAD_LIST || CC_PolicyAuthorizeNV)
837         (COMMAND_ATTRIBUTES)(CC_PolicyAuthorizeNV           *  // 0x0192
838             (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
839 #endif
840 #if (PAD_LIST || CC_EncryptDecrypt2)
841         (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt2             *  // 0x0193
842             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
843 #endif
844 #if (PAD_LIST || CC_AC_GetCapability)
```

```
845             (COMMAND_ATTRIBUTES)(CC_AC_GetCapability            *  // 0x0194
846                 (IS_IMPLEMENTED)),
847     #endif
848     #if (PAD_LIST || CC_AC_Send)
849             (COMMAND_ATTRIBUTES)(CC_AC_Send                     *  // 0x0195
850                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
851     #endif
852     #if (PAD_LIST || CC_Policy_AC_SendSelect)
853             (COMMAND_ATTRIBUTES)(CC_Policy_AC_SendSelect        *  // 0x0196
854                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
855     #endif
856     #if (PAD_LIST || CC_CertifyX509)
857             (COMMAND_ATTRIBUTES)(CC_CertifyX509                 *  // 0x0197
858                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
859     #endif
860     #if (PAD_LIST || CC_ACT_SetTimeout)
861             (COMMAND_ATTRIBUTES)(CC_ACT_SetTimeout              *  // 0x0198
862                 (IS_IMPLEMENTED+HANDLE_1_USER)),
863     #endif
864     #if (PAD_LIST || CC_Vendor_TCG_Test)
865             (COMMAND_ATTRIBUTES)(CC_Vendor_TCG_Test             *  // 0x0000
866                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
867     #endif
868             0
869     };
870     #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

```
#ifdef _COMMAND_TABLE_DISPATCH_
```

Define the stop value

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

These macros provide some variability in how the data is encoded. They also make the lines a little shorter. ;-)

```
#if TABLE_DRIVEN_MARSHAL
#   define UNMARSHAL_DISPATCH(name)    (marshalIndex_t)name##_MARSHAL_REF
#   define MARSHAL_DISPATCH(name)      (marshalIndex_t)name##_MARSHAL_REF
#   define _UNMARSHAL_T_     marshalIndex_t
#   define _MARSHAL_T_       marshalIndex_t
#
#else
#   define UNMARSHAL_DISPATCH(name)    (UNMARSHAL_t)name##_Unmarshal
#   define MARSHAL_DISPATCH(name)      (MARSHAL_t)name##_Marshal
#   define _UNMARSHAL_T_     UNMARSHAL_t
#   define _MARSHAL_T_       MARSHAL_t
#endif
```

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

```
const _UNMARSHAL_T_ UnmarshalArray[] = {
#define TPMI_DH_CONTEXT_H_UNMARSHAL             0
             UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
#define TPMI_RH_AC_H_UNMARSHAL                  (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_AC),
#define TPMI_RH_ACT_H_UNMARSHAL                 (TPMI_RH_AC_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_ACT),
#define TPMI_RH_CLEAR_H_UNMARSHAL               (TPMI_RH_ACT_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_CLEAR),
#define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL      (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_AUTH),
#define TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL    \
            (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_POLICY),
#define TPMI_RH_LOCKOUT_H_UNMARSHAL             \
            (TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_LOCKOUT),
#define TPMI_RH_NV_AUTH_H_UNMARSHAL             (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_NV_AUTH),
#define TPMI_RH_NV_INDEX_H_UNMARSHAL            (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_NV_INDEX),
#define TPMI_RH_PLATFORM_H_UNMARSHAL            (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_PLATFORM),
#define TPMI_RH_PROVISION_H_UNMARSHAL           (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_RH_PROVISION),
#define TPMI_SH_HMAC_H_UNMARSHAL                (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_SH_HMAC),
#define TPMI_SH_POLICY_H_UNMARSHAL              (TPMI_SH_HMAC_H_UNMARSHAL + 1)
            UNMARSHAL_DISPATCH(TPMI_SH_POLICY),
// HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
#define HANDLE_FIRST_FLAG_TYPE                  (TPMI_SH_POLICY_H_UNMARSHAL + 1)
```



```
47  #define TPMI_DH_ENTITY_H_UNMARSHAL              (TPMI_SH_POLICY_H_UNMARSHAL + 1)
48              UNMARSHAL_DISPATCH(TPMI_DH_ENTITY),
49  #define TPMI_DH_OBJECT_H_UNMARSHAL              (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
50              UNMARSHAL_DISPATCH(TPMI_DH_OBJECT),
51  #define TPMI_DH_PARENT_H_UNMARSHAL              (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
52              UNMARSHAL_DISPATCH(TPMI_DH_PARENT),
53  #define TPMI_DH_PCR_H_UNMARSHAL                 (TPMI_DH_PARENT_H_UNMARSHAL + 1)
54              UNMARSHAL_DISPATCH(TPMI_DH_PCR),
55  #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL         (TPMI_DH_PCR_H_UNMARSHAL + 1)
56              UNMARSHAL_DISPATCH(TPMI_RH_ENDORSEMENT),
57  #define TPMI_RH_HIERARCHY_H_UNMARSHAL           \
58              (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
59              UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
60  // PARAMETER_FIRST_TYPE marks the end of the handle list.
61  #define PARAMETER_FIRST_TYPE                    (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
62  #define TPM2B_DATA_P_UNMARSHAL                  (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
63              UNMARSHAL_DISPATCH(TPM2B_DATA),
64  #define TPM2B_DIGEST_P_UNMARSHAL                (TPM2B_DATA_P_UNMARSHAL + 1)
65              UNMARSHAL_DISPATCH(TPM2B_DIGEST),
66  #define TPM2B_ECC_PARAMETER_P_UNMARSHAL         (TPM2B_DIGEST_P_UNMARSHAL + 1)
67              UNMARSHAL_DISPATCH(TPM2B_ECC_PARAMETER),
68  #define TPM2B_ECC_POINT_P_UNMARSHAL             \
69              (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
70              UNMARSHAL_DISPATCH(TPM2B_ECC_POINT),
71  #define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL      (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
72              UNMARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
73  #define TPM2B_EVENT_P_UNMARSHAL                 \
74              (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
75              UNMARSHAL_DISPATCH(TPM2B_EVENT),
76  #define TPM2B_ID_OBJECT_P_UNMARSHAL             (TPM2B_EVENT_P_UNMARSHAL + 1)
77              UNMARSHAL_DISPATCH(TPM2B_ID_OBJECT),
78  #define TPM2B_IV_P_UNMARSHAL                    (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
79              UNMARSHAL_DISPATCH(TPM2B_IV),
80  #define TPM2B_MAX_BUFFER_P_UNMARSHAL            (TPM2B_IV_P_UNMARSHAL + 1)
81              UNMARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
82  #define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL         (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
83              UNMARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
84  #define TPM2B_NAME_P_UNMARSHAL                  \
85              (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
86              UNMARSHAL_DISPATCH(TPM2B_NAME),
87  #define TPM2B_NV_PUBLIC_P_UNMARSHAL             (TPM2B_NAME_P_UNMARSHAL + 1)
88              UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
89  #define TPM2B_PRIVATE_P_UNMARSHAL               (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
90              UNMARSHAL_DISPATCH(TPM2B_PRIVATE),
91  #define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL        (TPM2B_PRIVATE_P_UNMARSHAL + 1)
92              UNMARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
93  #define TPM2B_SENSITIVE_P_UNMARSHAL             \
94              (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
95              UNMARSHAL_DISPATCH(TPM2B_SENSITIVE),
96  #define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL      (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
97              UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_CREATE),
98  #define TPM2B_SENSITIVE_DATA_P_UNMARSHAL        \
99              (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
100             UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
101 #define TPM2B_TEMPLATE_P_UNMARSHAL              \
102             (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
103             UNMARSHAL_DISPATCH(TPM2B_TEMPLATE),
104 #define TPM2B_TIMEOUT_P_UNMARSHAL               (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
105             UNMARSHAL_DISPATCH(TPM2B_TIMEOUT),
106 #define TPMI_DH_CONTEXT_P_UNMARSHAL             (TPM2B_TIMEOUT_P_UNMARSHAL + 1)
107             UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
108 #define TPMI_DH_PERSISTENT_P_UNMARSHAL          (TPMI_DH_CONTEXT_P_UNMARSHAL + 1)
109             UNMARSHAL_DISPATCH(TPMI_DH_PERSISTENT),
110 #define TPMI_ECC_CURVE_P_UNMARSHAL              (TPMI_DH_PERSISTENT_P_UNMARSHAL + 1)
111             UNMARSHAL_DISPATCH(TPMI_ECC_CURVE),
112 #define TPMI_YES_NO_P_UNMARSHAL                 (TPMI_ECC_CURVE_P_UNMARSHAL + 1)
```

```
113             UNMARSHAL_DISPATCH(TPMI_YES_NO),
114 #define TPML_ALG_P_UNMARSHAL                    (TPMI_YES_NO_P_UNMARSHAL + 1)
115             UNMARSHAL_DISPATCH(TPML_ALG),
116 #define TPML_CC_P_UNMARSHAL                     (TPML_ALG_P_UNMARSHAL + 1)
117             UNMARSHAL_DISPATCH(TPML_CC),
118 #define TPML_DIGEST_P_UNMARSHAL                 (TPML_CC_P_UNMARSHAL + 1)
119             UNMARSHAL_DISPATCH(TPML_DIGEST),
120 #define TPML_DIGEST_VALUES_P_UNMARSHAL          (TPML_DIGEST_P_UNMARSHAL + 1)
121             UNMARSHAL_DISPATCH(TPML_DIGEST_VALUES),
122 #define TPML_PCR_SELECTION_P_UNMARSHAL          (TPML_DIGEST_VALUES_P_UNMARSHAL + 1)
123             UNMARSHAL_DISPATCH(TPML_PCR_SELECTION),
124 #define TPMS_CONTEXT_P_UNMARSHAL                (TPML_PCR_SELECTION_P_UNMARSHAL + 1)
125             UNMARSHAL_DISPATCH(TPMS_CONTEXT),
126 #define TPMT_PUBLIC_PARMS_P_UNMARSHAL           (TPMS_CONTEXT_P_UNMARSHAL + 1)
127             UNMARSHAL_DISPATCH(TPMT_PUBLIC_PARMS),
128 #define TPMT_TK_AUTH_P_UNMARSHAL                (TPMT_PUBLIC_PARMS_P_UNMARSHAL + 1)
129             UNMARSHAL_DISPATCH(TPMT_TK_AUTH),
130 #define TPMT_TK_CREATION_P_UNMARSHAL            (TPMT_TK_AUTH_P_UNMARSHAL + 1)
131             UNMARSHAL_DISPATCH(TPMT_TK_CREATION),
132 #define TPMT_TK_HASHCHECK_P_UNMARSHAL           (TPMT_TK_CREATION_P_UNMARSHAL + 1)
133             UNMARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
134 #define TPMT_TK_VERIFIED_P_UNMARSHAL            (TPMT_TK_HASHCHECK_P_UNMARSHAL + 1)
135             UNMARSHAL_DISPATCH(TPMT_TK_VERIFIED),
136 #define TPM_AT_P_UNMARSHAL                      (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
137             UNMARSHAL_DISPATCH(TPM_AT),
138 #define TPM_CAP_P_UNMARSHAL                     (TPM_AT_P_UNMARSHAL + 1)
139             UNMARSHAL_DISPATCH(TPM_CAP),
140 #define TPM_CLOCK_ADJUST_P_UNMARSHAL            (TPM_CAP_P_UNMARSHAL + 1)
141             UNMARSHAL_DISPATCH(TPM_CLOCK_ADJUST),
142 #define TPM_EO_P_UNMARSHAL                      (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
143             UNMARSHAL_DISPATCH(TPM_EO),
144 #define TPM_SE_P_UNMARSHAL                      (TPM_EO_P_UNMARSHAL + 1)
145             UNMARSHAL_DISPATCH(TPM_SE),
146 #define TPM_SU_P_UNMARSHAL                      (TPM_SE_P_UNMARSHAL + 1)
147             UNMARSHAL_DISPATCH(TPM_SU),
148 #define UINT16_P_UNMARSHAL                      (TPM_SU_P_UNMARSHAL + 1)
149             UNMARSHAL_DISPATCH(UINT16),
150 #define UINT32_P_UNMARSHAL                      (UINT16_P_UNMARSHAL + 1)
151             UNMARSHAL_DISPATCH(UINT32),
152 #define UINT64_P_UNMARSHAL                      (UINT32_P_UNMARSHAL + 1)
153             UNMARSHAL_DISPATCH(UINT64),
154 #define UINT8_P_UNMARSHAL                       (UINT64_P_UNMARSHAL + 1)
155             UNMARSHAL_DISPATCH(UINT8),
156 // PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
157 #define PARAMETER_FIRST_FLAG_TYPE               (UINT8_P_UNMARSHAL + 1)
158 #define TPM2B_PUBLIC_P_UNMARSHAL                (UINT8_P_UNMARSHAL + 1)
159             UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
160 #define TPMI_ALG_CIPHER_MODE_P_UNMARSHAL        (TPM2B_PUBLIC_P_UNMARSHAL + 1)
161             UNMARSHAL_DISPATCH(TPMI_ALG_CIPHER_MODE),
162 #define TPMI_ALG_HASH_P_UNMARSHAL               \
163             (TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + 1)
164             UNMARSHAL_DISPATCH(TPMI_ALG_HASH),
165 #define TPMI_ALG_MAC_SCHEME_P_UNMARSHAL         (TPMI_ALG_HASH_P_UNMARSHAL + 1)
166             UNMARSHAL_DISPATCH(TPMI_ALG_MAC_SCHEME),
167 #define TPMI_DH_PCR_P_UNMARSHAL                 \
168             (TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + 1)
169             UNMARSHAL_DISPATCH(TPMI_DH_PCR),
170 #define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL       (TPMI_DH_PCR_P_UNMARSHAL + 1)
171             UNMARSHAL_DISPATCH(TPMI_ECC_KEY_EXCHANGE),
172 #define TPMI_RH_ENABLES_P_UNMARSHAL             \
173             (TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
174             UNMARSHAL_DISPATCH(TPMI_RH_ENABLES),
175 #define TPMI_RH_HIERARCHY_P_UNMARSHAL           (TPMI_RH_ENABLES_P_UNMARSHAL + 1)
176             UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
177 #define TPMT_RSA_DECRYPT_P_UNMARSHAL            (TPMI_RH_HIERARCHY_P_UNMARSHAL + 1)
178             UNMARSHAL_DISPATCH(TPMT_RSA_DECRYPT),
```

```
179   #define TPMT_SIGNATURE_P_UNMARSHAL              (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
180               UNMARSHAL_DISPATCH(TPMT_SIGNATURE),
181   #define TPMT_SIG_SCHEME_P_UNMARSHAL             (TPMT_SIGNATURE_P_UNMARSHAL + 1)
182               UNMARSHAL_DISPATCH(TPMT_SIG_SCHEME),
183   #define TPMT_SYM_DEF_P_UNMARSHAL                (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
184               UNMARSHAL_DISPATCH(TPMT_SYM_DEF),
185   #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL         (TPMT_SYM_DEF_P_UNMARSHAL + 1)
186               UNMARSHAL_DISPATCH(TPMT_SYM_DEF_OBJECT)
187   // PARAMETER_LAST_TYPE is the end of the command parameter list.
188   #define PARAMETER_LAST_TYPE                     (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
189   };
```

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

```
190   const _MARSHAL_T_ MarshalArray[] = {
191
192   #define UINT32_H_MARSHAL                        0
193               MARSHAL_DISPATCH(UINT32),
194   // RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
195   #define RESPONSE_PARAMETER_FIRST_TYPE           (UINT32_H_MARSHAL + 1)
196   #define TPM2B_ATTEST_P_MARSHAL                  (UINT32_H_MARSHAL + 1)
197               MARSHAL_DISPATCH(TPM2B_ATTEST),
198   #define TPM2B_CREATION_DATA_P_MARSHAL           (TPM2B_ATTEST_P_MARSHAL + 1)
199               MARSHAL_DISPATCH(TPM2B_CREATION_DATA),
200   #define TPM2B_DATA_P_MARSHAL                    (TPM2B_CREATION_DATA_P_MARSHAL + 1)
201               MARSHAL_DISPATCH(TPM2B_DATA),
202   #define TPM2B_DIGEST_P_MARSHAL                  (TPM2B_DATA_P_MARSHAL + 1)
203               MARSHAL_DISPATCH(TPM2B_DIGEST),
204   #define TPM2B_ECC_POINT_P_MARSHAL               (TPM2B_DIGEST_P_MARSHAL + 1)
205               MARSHAL_DISPATCH(TPM2B_ECC_POINT),
206   #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL        (TPM2B_ECC_POINT_P_MARSHAL + 1)
207               MARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
208   #define TPM2B_ID_OBJECT_P_MARSHAL               \
209               (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
210               MARSHAL_DISPATCH(TPM2B_ID_OBJECT),
211   #define TPM2B_IV_P_MARSHAL                      (TPM2B_ID_OBJECT_P_MARSHAL + 1)
212               MARSHAL_DISPATCH(TPM2B_IV),
213   #define TPM2B_MAX_BUFFER_P_MARSHAL              (TPM2B_IV_P_MARSHAL + 1)
214               MARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
215   #define TPM2B_MAX_NV_BUFFER_P_MARSHAL           (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
216               MARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
217   #define TPM2B_NAME_P_MARSHAL                    (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
218               MARSHAL_DISPATCH(TPM2B_NAME),
219   #define TPM2B_NV_PUBLIC_P_MARSHAL               (TPM2B_NAME_P_MARSHAL + 1)
220               MARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
221   #define TPM2B_PRIVATE_P_MARSHAL                 (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
222               MARSHAL_DISPATCH(TPM2B_PRIVATE),
223   #define TPM2B_PUBLIC_P_MARSHAL                  (TPM2B_PRIVATE_P_MARSHAL + 1)
224               MARSHAL_DISPATCH(TPM2B_PUBLIC),
225   #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL          (TPM2B_PUBLIC_P_MARSHAL + 1)
226               MARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
227   #define TPM2B_SENSITIVE_DATA_P_MARSHAL          (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
228               MARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
229   #define TPM2B_TIMEOUT_P_MARSHAL                 (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
230               MARSHAL_DISPATCH(TPM2B_TIMEOUT),
231   #define UINT8_P_MARSHAL                         (TPM2B_TIMEOUT_P_MARSHAL + 1)
232               MARSHAL_DISPATCH(UINT8),
233   #define TPML_AC_CAPABILITIES_P_MARSHAL          (UINT8_P_MARSHAL + 1)
234               MARSHAL_DISPATCH(TPML_AC_CAPABILITIES),
235   #define TPML_ALG_P_MARSHAL                      (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
236               MARSHAL_DISPATCH(TPML_ALG),
237   #define TPML_DIGEST_P_MARSHAL                   (TPML_ALG_P_MARSHAL + 1)
238               MARSHAL_DISPATCH(TPML_DIGEST),
239   #define TPML_DIGEST_VALUES_P_MARSHAL            (TPML_DIGEST_P_MARSHAL + 1)
```

```
240             MARSHAL_DISPATCH(TPML_DIGEST_VALUES),
241     #define TPML_PCR_SELECTION_P_MARSHAL            (TPML_DIGEST_VALUES_P_MARSHAL + 1)
242             MARSHAL_DISPATCH(TPML_PCR_SELECTION),
243     #define TPMS_AC_OUTPUT_P_MARSHAL                (TPML_PCR_SELECTION_P_MARSHAL + 1)
244             MARSHAL_DISPATCH(TPMS_AC_OUTPUT),
245     #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL     (TPMS_AC_OUTPUT_P_MARSHAL + 1)
246             MARSHAL_DISPATCH(TPMS_ALGORITHM_DETAIL_ECC),
247     #define TPMS_CAPABILITY_DATA_P_MARSHAL          \
248             (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
249             MARSHAL_DISPATCH(TPMS_CAPABILITY_DATA),
250     #define TPMS_CONTEXT_P_MARSHAL                  (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
251             MARSHAL_DISPATCH(TPMS_CONTEXT),
252     #define TPMS_TIME_INFO_P_MARSHAL                (TPMS_CONTEXT_P_MARSHAL + 1)
253             MARSHAL_DISPATCH(TPMS_TIME_INFO),
254     #define TPMT_HA_P_MARSHAL                       (TPMS_TIME_INFO_P_MARSHAL + 1)
255             MARSHAL_DISPATCH(TPMT_HA),
256     #define TPMT_SIGNATURE_P_MARSHAL                (TPMT_HA_P_MARSHAL + 1)
257             MARSHAL_DISPATCH(TPMT_SIGNATURE),
258     #define TPMT_TK_AUTH_P_MARSHAL                  (TPMT_SIGNATURE_P_MARSHAL + 1)
259             MARSHAL_DISPATCH(TPMT_TK_AUTH),
260     #define TPMT_TK_CREATION_P_MARSHAL              (TPMT_TK_AUTH_P_MARSHAL + 1)
261             MARSHAL_DISPATCH(TPMT_TK_CREATION),
262     #define TPMT_TK_HASHCHECK_P_MARSHAL             (TPMT_TK_CREATION_P_MARSHAL + 1)
263             MARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
264     #define TPMT_TK_VERIFIED_P_MARSHAL              (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
265             MARSHAL_DISPATCH(TPMT_TK_VERIFIED),
266     #define UINT32_P_MARSHAL                        (TPMT_TK_VERIFIED_P_MARSHAL + 1)
267             MARSHAL_DISPATCH(UINT32),
268     #define UINT16_P_MARSHAL                        (UINT32_P_MARSHAL + 1)
269             MARSHAL_DISPATCH(UINT16)
270     // RESPONSE_PARAMETER_LAST_TYPE is the end of the response parameter list.
271     #define RESPONSE_PARAMETER_LAST_TYPE            (UINT16_P_MARSHAL)
272     };
```

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

```
273     #define INT32_P_UNMARSHAL                       UINT32_P_UNMARSHAL
274     #define TPM2B_AUTH_P_UNMARSHAL                  TPM2B_DIGEST_P_UNMARSHAL
275     #define TPM2B_NONCE_P_UNMARSHAL                 TPM2B_DIGEST_P_UNMARSHAL
276     #define TPM2B_OPERAND_P_UNMARSHAL               TPM2B_DIGEST_P_UNMARSHAL
277     #define TPMA_LOCALITY_P_UNMARSHAL               UINT8_P_UNMARSHAL
278     #define TPM_CC_P_UNMARSHAL                      UINT32_P_UNMARSHAL
279     #define TPMI_DH_CONTEXT_H_MARSHAL               UINT32_H_MARSHAL
280     #define TPMI_DH_OBJECT_H_MARSHAL                UINT32_H_MARSHAL
281     #define TPMI_SH_AUTH_SESSION_H_MARSHAL          UINT32_H_MARSHAL
282     #define TPM_HANDLE_H_MARSHAL                    UINT32_H_MARSHAL
283     #define TPM2B_NONCE_P_MARSHAL                   TPM2B_DIGEST_P_MARSHAL
284     #define TPMI_YES_NO_P_MARSHAL                   UINT8_P_MARSHAL
285     #define TPM_RC_P_MARSHAL                        UINT32_P_MARSHAL
286     #if CC_Startup
287     #include "Startup_fp.h"
288     typedef TPM_RC  (Startup_Entry)(
289         Startup_In                  *in
290     );
291     typedef const struct {
292         Startup_Entry           *entry;
293         UINT16                  inSize;
294         UINT16                  outSize;
295         UINT16                  offsetOfTypes;
296         BYTE                    types[3];
297     } Startup_COMMAND_DESCRIPTOR_t;
298     Startup_COMMAND_DESCRIPTOR_t _StartupData = {
299         /* entry         */     &TPM2_Startup,
300         /* inSize        */     (UINT16)(sizeof(Startup_In)),
```

```
301         /* outSize       */     0,
302         /* offsetOfTypes */     offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
303         /* offsets       */     // No parameter offsets;
304         /* types         */     {TPM_SU_P_UNMARSHAL,
305                                   END_OF_LIST,
306                                   END_OF_LIST}
307     };
308     #define _StartupDataAddress (&_StartupData)
309     #else
310     #define _StartupDataAddress 0
311     #endif // CC_Startup
312     #if CC_Shutdown
313     #include "Shutdown_fp.h"
314     typedef TPM_RC  (Shutdown_Entry)(
315         Shutdown_In                 *in
316     );
317     typedef const struct {
318         Shutdown_Entry          *entry;
319         UINT16                  inSize;
320         UINT16                  outSize;
321         UINT16                  offsetOfTypes;
322         BYTE                    types[3];
323     } Shutdown_COMMAND_DESCRIPTOR_t;
324     Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
325         /* entry         */     &TPM2_Shutdown,
326         /* inSize        */     (UINT16)(sizeof(Shutdown_In)),
327         /* outSize       */     0,
328         /* offsetOfTypes */     offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
329         /* offsets       */     // No parameter offsets;
330         /* types         */     {TPM_SU_P_UNMARSHAL,
331                                   END_OF_LIST,
332                                   END_OF_LIST}
333     };
334     #define _ShutdownDataAddress (&_ShutdownData)
335     #else
336     #define _ShutdownDataAddress 0
337     #endif // CC_Shutdown
338     #if CC_SelfTest
339     #include "SelfTest_fp.h"
340     typedef TPM_RC  (SelfTest_Entry)(
341         SelfTest_In                 *in
342     );
343     typedef const struct {
344         SelfTest_Entry          *entry;
345         UINT16                  inSize;
346         UINT16                  outSize;
347         UINT16                  offsetOfTypes;
348         BYTE                    types[3];
349     } SelfTest_COMMAND_DESCRIPTOR_t;
350     SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
351         /* entry         */     &TPM2_SelfTest,
352         /* inSize        */     (UINT16)(sizeof(SelfTest_In)),
353         /* outSize       */     0,
354         /* offsetOfTypes */     offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
355         /* offsets       */     // No parameter offsets;
356         /* types         */     {TPMI_YES_NO_P_UNMARSHAL,
357                                   END_OF_LIST,
358                                   END_OF_LIST}
359     };
360     #define _SelfTestDataAddress (&_SelfTestData)
361     #else
362     #define _SelfTestDataAddress 0
363     #endif // CC_SelfTest
364     #if CC_IncrementalSelfTest
365     #include "IncrementalSelfTest_fp.h"
366     typedef TPM_RC  (IncrementalSelfTest_Entry)(
```

```
367         IncrementalSelfTest_In          *in,
368         IncrementalSelfTest_Out         *out
369     );
370     typedef const struct {
371         IncrementalSelfTest_Entry   *entry;
372         UINT16                      inSize;
373         UINT16                      outSize;
374         UINT16                      offsetOfTypes;
375         BYTE                        types[4];
376     } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
377     IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
378         /* entry         */         &TPM2_IncrementalSelfTest,
379         /* inSize        */         (UINT16)(sizeof(IncrementalSelfTest_In)),
380         /* outSize       */         (UINT16)(sizeof(IncrementalSelfTest_Out)),
381         /* offsetOfTypes */         offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t,
    types),
382         /* offsets       */         // No parameter offsets;
383         /* types         */         {TPML_ALG_P_UNMARSHAL,
384                                      END_OF_LIST,
385                                      TPML_ALG_P_MARSHAL,
386                                      END_OF_LIST}
387     };
388     #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
389     #else
390     #define _IncrementalSelfTestDataAddress 0
391     #endif // CC_IncrementalSelfTest
392     #if CC_GetTestResult
393     #include "GetTestResult_fp.h"
394     typedef TPM_RC  (GetTestResult_Entry)(
395         GetTestResult_Out           *out
396     );
397     typedef const struct {
398         GetTestResult_Entry     *entry;
399         UINT16                  inSize;
400         UINT16                  outSize;
401         UINT16                  offsetOfTypes;
402         UINT16                  paramOffsets[1];
403         BYTE                    types[4];
404     } GetTestResult_COMMAND_DESCRIPTOR_t;
405     GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
406         /* entry         */     &TPM2_GetTestResult,
407         /* inSize        */     0,
408         /* outSize       */     (UINT16)(sizeof(GetTestResult_Out)),
409         /* offsetOfTypes */     offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
410         /* offsets       */     {(UINT16)(offsetof(GetTestResult_Out, testResult))},
411         /* types         */     {END_OF_LIST,
412                                  TPM2B_MAX_BUFFER_P_MARSHAL,
413                                  TPM_RC_P_MARSHAL,
414                                  END_OF_LIST}
415     };
416     #define _GetTestResultDataAddress (&_GetTestResultData)
417     #else
418     #define _GetTestResultDataAddress 0
419     #endif // CC_GetTestResult
420     #if CC_StartAuthSession
421     #include "StartAuthSession_fp.h"
422     typedef TPM_RC  (StartAuthSession_Entry)(
423         StartAuthSession_In         *in,
424         StartAuthSession_Out        *out
425     );
426     typedef const struct {
427         StartAuthSession_Entry  *entry;
428         UINT16                  inSize;
429         UINT16                  outSize;
430         UINT16                  offsetOfTypes;
431         UINT16                  paramOffsets[7];
```

```
    BYTE                    types[11];
} StartAuthSession_COMMAND_DESCRIPTOR_t;
StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
    /* entry         */     &TPM2_StartAuthSession,
    /* inSize        */     (UINT16)(sizeof(StartAuthSession_In)),
    /* outSize       */     (UINT16)(sizeof(StartAuthSession_Out)),
    /* offsetOfTypes */     offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(StartAuthSession_In, bind)),
                             (UINT16)(offsetof(StartAuthSession_In, nonceCaller)),
                             (UINT16)(offsetof(StartAuthSession_In, encryptedSalt)),
                             (UINT16)(offsetof(StartAuthSession_In, sessionType)),
                             (UINT16)(offsetof(StartAuthSession_In, symmetric)),
                             (UINT16)(offsetof(StartAuthSession_In, authHash)),
                             (UINT16)(offsetof(StartAuthSession_Out, nonceTPM))},
    /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
                             TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
                             TPM2B_NONCE_P_UNMARSHAL,
                             TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
                             TPM_SE_P_UNMARSHAL,
                             TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
                             TPMI_ALG_HASH_P_UNMARSHAL,
                             END_OF_LIST,
                             TPMI_SH_AUTH_SESSION_H_MARSHAL,
                             TPM2B_NONCE_P_MARSHAL,
                             END_OF_LIST}
};
#define _StartAuthSessionDataAddress (&_StartAuthSessionData)
#else
#define _StartAuthSessionDataAddress 0
#endif // CC_StartAuthSession
#if CC_PolicyRestart
#include "PolicyRestart_fp.h"
typedef TPM_RC  (PolicyRestart_Entry)(
    PolicyRestart_In            *in
);
typedef const struct {
    PolicyRestart_Entry     *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    BYTE                    types[3];
} PolicyRestart_COMMAND_DESCRIPTOR_t;
PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
    /* entry         */     &TPM2_PolicyRestart,
    /* inSize        */     (UINT16)(sizeof(PolicyRestart_In)),
    /* outSize       */     0,
    /* offsetOfTypes */     offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     // No parameter offsets;
    /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
                             END_OF_LIST,
                             END_OF_LIST}
};
#define _PolicyRestartDataAddress (&_PolicyRestartData)
#else
#define _PolicyRestartDataAddress 0
#endif // CC_PolicyRestart
#if CC_Create
#include "Create_fp.h"
typedef TPM_RC  (Create_Entry)(
    Create_In                   *in,
    Create_Out                  *out
);
typedef const struct {
    Create_Entry            *entry;
    UINT16                  inSize;
    UINT16                  outSize;
```

```
498         UINT16                  offsetOfTypes;
499         UINT16                  paramOffsets[8];
500         BYTE                    types[12];
501     } Create_COMMAND_DESCRIPTOR_t;
502     Create_COMMAND_DESCRIPTOR_t _CreateData = {
503         /* entry         */     &TPM2_Create,
504         /* inSize        */     (UINT16)(sizeof(Create_In)),
505         /* outSize       */     (UINT16)(sizeof(Create_Out)),
506         /* offsetOfTypes */     offsetof(Create_COMMAND_DESCRIPTOR_t, types),
507         /* offsets       */     {(UINT16)(offsetof(Create_In, inSensitive)),
508                                  (UINT16)(offsetof(Create_In, inPublic)),
509                                  (UINT16)(offsetof(Create_In, outsideInfo)),
510                                  (UINT16)(offsetof(Create_In, creationPCR)),
511                                  (UINT16)(offsetof(Create_Out, outPublic)),
512                                  (UINT16)(offsetof(Create_Out, creationData)),
513                                  (UINT16)(offsetof(Create_Out, creationHash)),
514                                  (UINT16)(offsetof(Create_Out, creationTicket))},
515         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
516                                  TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
517                                  TPM2B_PUBLIC_P_UNMARSHAL,
518                                  TPM2B_DATA_P_UNMARSHAL,
519                                  TPML_PCR_SELECTION_P_UNMARSHAL,
520                                  END_OF_LIST,
521                                  TPM2B_PRIVATE_P_MARSHAL,
522                                  TPM2B_PUBLIC_P_MARSHAL,
523                                  TPM2B_CREATION_DATA_P_MARSHAL,
524                                  TPM2B_DIGEST_P_MARSHAL,
525                                  TPMT_TK_CREATION_P_MARSHAL,
526                                  END_OF_LIST}
527     };
528     #define _CreateDataAddress (&_CreateData)
529     #else
530     #define _CreateDataAddress 0
531     #endif // CC_Create
532     #if CC_Load
533     #include "Load_fp.h"
534     typedef TPM_RC  (Load_Entry)(
535         Load_In                     *in,
536         Load_Out                    *out
537     );
538     typedef const struct {
539         Load_Entry              *entry;
540         UINT16                  inSize;
541         UINT16                  outSize;
542         UINT16                  offsetOfTypes;
543         UINT16                  paramOffsets[3];
544         BYTE                    types[7];
545     } Load_COMMAND_DESCRIPTOR_t;
546     Load_COMMAND_DESCRIPTOR_t _LoadData = {
547         /* entry         */     &TPM2_Load,
548         /* inSize        */     (UINT16)(sizeof(Load_In)),
549         /* outSize       */     (UINT16)(sizeof(Load_Out)),
550         /* offsetOfTypes */     offsetof(Load_COMMAND_DESCRIPTOR_t, types),
551         /* offsets       */     {(UINT16)(offsetof(Load_In, inPrivate)),
552                                  (UINT16)(offsetof(Load_In, inPublic)),
553                                  (UINT16)(offsetof(Load_Out, name))},
554         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
555                                  TPM2B_PRIVATE_P_UNMARSHAL,
556                                  TPM2B_PUBLIC_P_UNMARSHAL,
557                                  END_OF_LIST,
558                                  TPM_HANDLE_H_MARSHAL,
559                                  TPM2B_NAME_P_MARSHAL,
560                                  END_OF_LIST}
561     };
562     #define _LoadDataAddress (&_LoadData)
563     #else
```

```
564  #define _LoadDataAddress 0
565  #endif // CC_Load
566  #if CC_LoadExternal
567  #include "LoadExternal_fp.h"
568  typedef TPM_RC  (LoadExternal_Entry)(
569      LoadExternal_In             *in,
570      LoadExternal_Out            *out
571  );
572  typedef const struct {
573      LoadExternal_Entry      *entry;
574      UINT16                  inSize;
575      UINT16                  outSize;
576      UINT16                  offsetOfTypes;
577      UINT16                  paramOffsets[3];
578      BYTE                    types[7];
579  } LoadExternal_COMMAND_DESCRIPTOR_t;
580  LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
581      /* entry         */     &TPM2_LoadExternal,
582      /* inSize        */     (UINT16)(sizeof(LoadExternal_In)),
583      /* outSize       */     (UINT16)(sizeof(LoadExternal_Out)),
584      /* offsetOfTypes */     offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
585      /* offsets       */     {(UINT16)(offsetof(LoadExternal_In, inPublic)),
586                               (UINT16)(offsetof(LoadExternal_In, hierarchy)),
587                               (UINT16)(offsetof(LoadExternal_Out, name))},
588      /* types         */     {TPM2B_SENSITIVE_P_UNMARSHAL,
589                               TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,
590                               TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
591                               END_OF_LIST,
592                               TPM_HANDLE_H_MARSHAL,
593                               TPM2B_NAME_P_MARSHAL,
594                               END_OF_LIST}
595  };
596  #define _LoadExternalDataAddress (&_LoadExternalData)
597  #else
598  #define _LoadExternalDataAddress 0
599  #endif // CC_LoadExternal
600  #if CC_ReadPublic
601  #include "ReadPublic_fp.h"
602  typedef TPM_RC  (ReadPublic_Entry)(
603      ReadPublic_In               *in,
604      ReadPublic_Out              *out
605  );
606  typedef const struct {
607      ReadPublic_Entry        *entry;
608      UINT16                  inSize;
609      UINT16                  outSize;
610      UINT16                  offsetOfTypes;
611      UINT16                  paramOffsets[2];
612      BYTE                    types[6];
613  } ReadPublic_COMMAND_DESCRIPTOR_t;
614  ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
615      /* entry         */     &TPM2_ReadPublic,
616      /* inSize        */     (UINT16)(sizeof(ReadPublic_In)),
617      /* outSize       */     (UINT16)(sizeof(ReadPublic_Out)),
618      /* offsetOfTypes */     offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
619      /* offsets       */     {(UINT16)(offsetof(ReadPublic_Out, name)),
620                               (UINT16)(offsetof(ReadPublic_Out, qualifiedName))},
621      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
622                               END_OF_LIST,
623                               TPM2B_PUBLIC_P_MARSHAL,
624                               TPM2B_NAME_P_MARSHAL,
625                               TPM2B_NAME_P_MARSHAL,
626                               END_OF_LIST}
627  };
628  #define _ReadPublicDataAddress (&_ReadPublicData)
629  #else
```

```
630 #define _ReadPublicDataAddress 0
631 #endif // CC_ReadPublic
632 #if CC_ActivateCredential
633 #include "ActivateCredential_fp.h"
634 typedef TPM_RC  (ActivateCredential_Entry)(
635     ActivateCredential_In           *in,
636     ActivateCredential_Out          *out
637 );
638 typedef const struct {
639     ActivateCredential_Entry    *entry;
640     UINT16                      inSize;
641     UINT16                      outSize;
642     UINT16                      offsetOfTypes;
643     UINT16                      paramOffsets[3];
644     BYTE                        types[7];
645 } ActivateCredential_COMMAND_DESCRIPTOR_t;
646 ActivateCredential_COMMAND_DESCRIPTOR_t _ActivateCredentialData = {
647     /* entry         */         &TPM2_ActivateCredential,
648     /* inSize        */         (UINT16)(sizeof(ActivateCredential_In)),
649     /* outSize       */         (UINT16)(sizeof(ActivateCredential_Out)),
650     /* offsetOfTypes */         offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t,
    types),
651     /* offsets       */         {(UINT16)(offsetof(ActivateCredential_In, keyHandle)),
652                                  (UINT16)(offsetof(ActivateCredential_In,
    credentialBlob)),
653                                  (UINT16)(offsetof(ActivateCredential_In, secret))},
654     /* types         */         {TPMI_DH_OBJECT_H_UNMARSHAL,
655                                  TPMI_DH_OBJECT_H_UNMARSHAL,
656                                  TPM2B_ID_OBJECT_P_UNMARSHAL,
657                                  TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
658                                  END_OF_LIST,
659                                  TPM2B_DIGEST_P_MARSHAL,
660                                  END_OF_LIST}
661 };
662 #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
663 #else
664 #define _ActivateCredentialDataAddress 0
665 #endif // CC_ActivateCredential
666 #if CC_MakeCredential
667 #include "MakeCredential_fp.h"
668 typedef TPM_RC  (MakeCredential_Entry)(
669     MakeCredential_In           *in,
670     MakeCredential_Out          *out
671 );
672 typedef const struct {
673     MakeCredential_Entry    *entry;
674     UINT16                  inSize;
675     UINT16                  outSize;
676     UINT16                  offsetOfTypes;
677     UINT16                  paramOffsets[3];
678     BYTE                    types[7];
679 } MakeCredential_COMMAND_DESCRIPTOR_t;
680 MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
681     /* entry         */     &TPM2_MakeCredential,
682     /* inSize        */     (UINT16)(sizeof(MakeCredential_In)),
683     /* outSize       */     (UINT16)(sizeof(MakeCredential_Out)),
684     /* offsetOfTypes */     offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
685     /* offsets       */     {(UINT16)(offsetof(MakeCredential_In, credential)),
686                              (UINT16)(offsetof(MakeCredential_In, objectName)),
687                              (UINT16)(offsetof(MakeCredential_Out, secret))},
688     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
689                              TPM2B_DIGEST_P_UNMARSHAL,
690                              TPM2B_NAME_P_UNMARSHAL,
691                              END_OF_LIST,
692                              TPM2B_ID_OBJECT_P_MARSHAL,
693                              TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
```

```
694                                 END_OF_LIST}
695     };
696     #define _MakeCredentialDataAddress (&_MakeCredentialData)
697     #else
698     #define _MakeCredentialDataAddress 0
699     #endif // CC_MakeCredential
700     #if CC_Unseal
701     #include "Unseal_fp.h"
702     typedef TPM_RC  (Unseal_Entry)(
703         Unseal_In                   *in,
704         Unseal_Out                  *out
705     );
706     typedef const struct {
707         Unseal_Entry            *entry;
708         UINT16                  inSize;
709         UINT16                  outSize;
710         UINT16                  offsetOfTypes;
711         BYTE                    types[4];
712     } Unseal_COMMAND_DESCRIPTOR_t;
713     Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
714         /* entry         */     &TPM2_Unseal,
715         /* inSize        */     (UINT16)(sizeof(Unseal_In)),
716         /* outSize       */     (UINT16)(sizeof(Unseal_Out)),
717         /* offsetOfTypes */     offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
718         /* offsets       */     // No parameter offsets;
719         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
720                                  END_OF_LIST,
721                                  TPM2B_SENSITIVE_DATA_P_MARSHAL,
722                                  END_OF_LIST}
723     };
724     #define _UnsealDataAddress (&_UnsealData)
725     #else
726     #define _UnsealDataAddress 0
727     #endif // CC_Unseal
728     #if CC_ObjectChangeAuth
729     #include "ObjectChangeAuth_fp.h"
730     typedef TPM_RC  (ObjectChangeAuth_Entry)(
731         ObjectChangeAuth_In         *in,
732         ObjectChangeAuth_Out        *out
733     );
734     typedef const struct {
735         ObjectChangeAuth_Entry  *entry;
736         UINT16                  inSize;
737         UINT16                  outSize;
738         UINT16                  offsetOfTypes;
739         UINT16                  paramOffsets[2];
740         BYTE                    types[6];
741     } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
742     ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
743         /* entry         */     &TPM2_ObjectChangeAuth,
744         /* inSize        */     (UINT16)(sizeof(ObjectChangeAuth_In)),
745         /* outSize       */     (UINT16)(sizeof(ObjectChangeAuth_Out)),
746         /* offsetOfTypes */     offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
747         /* offsets       */     {(UINT16)(offsetof(ObjectChangeAuth_In, parentHandle)),
748                                   (UINT16)(offsetof(ObjectChangeAuth_In, newAuth))},
749         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
750                                  TPMI_DH_OBJECT_H_UNMARSHAL,
751                                  TPM2B_AUTH_P_UNMARSHAL,
752                                  END_OF_LIST,
753                                  TPM2B_PRIVATE_P_MARSHAL,
754                                  END_OF_LIST}
755     };
756     #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
757     #else
758     #define _ObjectChangeAuthDataAddress 0
759     #endif // CC_ObjectChangeAuth
```

```
760 #if CC_CreateLoaded
761 #include "CreateLoaded_fp.h"
762 typedef TPM_RC  (CreateLoaded_Entry)(
763     CreateLoaded_In             *in,
764     CreateLoaded_Out            *out
765 );
766 typedef const struct {
767     CreateLoaded_Entry      *entry;
768     UINT16                  inSize;
769     UINT16                  outSize;
770     UINT16                  offsetOfTypes;
771     UINT16                  paramOffsets[5];
772     BYTE                    types[9];
773 } CreateLoaded_COMMAND_DESCRIPTOR_t;
774 CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
775     /* entry         */     &TPM2_CreateLoaded,
776     /* inSize        */     (UINT16)(sizeof(CreateLoaded_In)),
777     /* outSize       */     (UINT16)(sizeof(CreateLoaded_Out)),
778     /* offsetOfTypes */     offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
779     /* offsets       */     {(UINT16)(offsetof(CreateLoaded_In, inSensitive)),
780                             (UINT16)(offsetof(CreateLoaded_In, inPublic)),
781                             (UINT16)(offsetof(CreateLoaded_Out, outPrivate)),
782                             (UINT16)(offsetof(CreateLoaded_Out, outPublic)),
783                             (UINT16)(offsetof(CreateLoaded_Out, name))},
784     /* types         */     {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
785                             TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
786                             TPM2B_TEMPLATE_P_UNMARSHAL,
787                             END_OF_LIST,
788                             TPM_HANDLE_H_MARSHAL,
789                             TPM2B_PRIVATE_P_MARSHAL,
790                             TPM2B_PUBLIC_P_MARSHAL,
791                             TPM2B_NAME_P_MARSHAL,
792                             END_OF_LIST}
793 };
794 #define _CreateLoadedDataAddress (&_CreateLoadedData)
795 #else
796 #define _CreateLoadedDataAddress 0
797 #endif // CC_CreateLoaded
798 #if CC_Duplicate
799 #include "Duplicate_fp.h"
800 typedef TPM_RC  (Duplicate_Entry)(
801     Duplicate_In                *in,
802     Duplicate_Out               *out
803 );
804 typedef const struct {
805     Duplicate_Entry         *entry;
806     UINT16                  inSize;
807     UINT16                  outSize;
808     UINT16                  offsetOfTypes;
809     UINT16                  paramOffsets[5];
810     BYTE                    types[9];
811 } Duplicate_COMMAND_DESCRIPTOR_t;
812 Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
813     /* entry         */     &TPM2_Duplicate,
814     /* inSize        */     (UINT16)(sizeof(Duplicate_In)),
815     /* outSize       */     (UINT16)(sizeof(Duplicate_Out)),
816     /* offsetOfTypes */     offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
817     /* offsets       */     {(UINT16)(offsetof(Duplicate_In, newParentHandle)),
818                             (UINT16)(offsetof(Duplicate_In, encryptionKeyIn)),
819                             (UINT16)(offsetof(Duplicate_In, symmetricAlg)),
820                             (UINT16)(offsetof(Duplicate_Out, duplicate)),
821                             (UINT16)(offsetof(Duplicate_Out, outSymSeed))},
822     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
823                             TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
824                             TPM2B_DATA_P_UNMARSHAL,
825                             TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
```

```
826                                 END_OF_LIST,
827                                 TPM2B_DATA_P_MARSHAL,
828                                 TPM2B_PRIVATE_P_MARSHAL,
829                                 TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
830                                 END_OF_LIST}
831 };
832 #define _DuplicateDataAddress (&_DuplicateData)
833 #else
834 #define _DuplicateDataAddress 0
835 #endif // CC_Duplicate
836 #if CC_Rewrap
837 #include "Rewrap_fp.h"
838 typedef TPM_RC  (Rewrap_Entry)(
839     Rewrap_In                   *in,
840     Rewrap_Out                  *out
841 );
842 typedef const struct {
843     Rewrap_Entry            *entry;
844     UINT16                  inSize;
845     UINT16                  outSize;
846     UINT16                  offsetOfTypes;
847     UINT16                  paramOffsets[5];
848     BYTE                    types[9];
849 } Rewrap_COMMAND_DESCRIPTOR_t;
850 Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {
851     /* entry         */     &TPM2_Rewrap,
852     /* inSize        */     (UINT16)(sizeof(Rewrap_In)),
853     /* outSize       */     (UINT16)(sizeof(Rewrap_Out)),
854     /* offsetOfTypes */     offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
855     /* offsets       */     {(UINT16)(offsetof(Rewrap_In, newParent)),
856                             (UINT16)(offsetof(Rewrap_In, inDuplicate)),
857                             (UINT16)(offsetof(Rewrap_In, name)),
858                             (UINT16)(offsetof(Rewrap_In, inSymSeed)),
859                             (UINT16)(offsetof(Rewrap_Out, outSymSeed))},
860     /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
861                             TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
862                             TPM2B_PRIVATE_P_UNMARSHAL,
863                             TPM2B_NAME_P_UNMARSHAL,
864                             TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
865                             END_OF_LIST,
866                             TPM2B_PRIVATE_P_MARSHAL,
867                             TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
868                             END_OF_LIST}
869 };
870 #define _RewrapDataAddress (&_RewrapData)
871 #else
872 #define _RewrapDataAddress 0
873 #endif // CC_Rewrap
874 #if CC_Import
875 #include "Import_fp.h"
876 typedef TPM_RC  (Import_Entry)(
877     Import_In                   *in,
878     Import_Out                  *out
879 );
880 typedef const struct {
881     Import_Entry            *entry;
882     UINT16                  inSize;
883     UINT16                  outSize;
884     UINT16                  offsetOfTypes;
885     UINT16                  paramOffsets[5];
886     BYTE                    types[9];
887 } Import_COMMAND_DESCRIPTOR_t;
888 Import_COMMAND_DESCRIPTOR_t _ImportData = {
889     /* entry         */     &TPM2_Import,
890     /* inSize        */     (UINT16)(sizeof(Import_In)),
891     /* outSize       */     (UINT16)(sizeof(Import_Out)),
```

```
892         /* offsetOfTypes */     offsetof(Import_COMMAND_DESCRIPTOR_t, types),
893         /* offsets       */     {(UINT16)(offsetof(Import_In, encryptionKey)),
894                                   (UINT16)(offsetof(Import_In, objectPublic)),
895                                   (UINT16)(offsetof(Import_In, duplicate)),
896                                   (UINT16)(offsetof(Import_In, inSymSeed)),
897                                   (UINT16)(offsetof(Import_In, symmetricAlg))},
898         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
899                                   TPM2B_DATA_P_UNMARSHAL,
900                                   TPM2B_PUBLIC_P_UNMARSHAL,
901                                   TPM2B_PRIVATE_P_UNMARSHAL,
902                                   TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
903                                   TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
904                                   END_OF_LIST,
905                                   TPM2B_PRIVATE_P_MARSHAL,
906                                   END_OF_LIST}
907     };
908     #define _ImportDataAddress (&_ImportData)
909     #else
910     #define _ImportDataAddress 0
911     #endif // CC_Import
912     #if CC_RSA_Encrypt
913     #include "RSA_Encrypt_fp.h"
914     typedef TPM_RC  (RSA_Encrypt_Entry)(
915         RSA_Encrypt_In              *in,
916         RSA_Encrypt_Out             *out
917     );
918     typedef const struct {
919         RSA_Encrypt_Entry       *entry;
920         UINT16                  inSize;
921         UINT16                  outSize;
922         UINT16                  offsetOfTypes;
923         UINT16                  paramOffsets[3];
924         BYTE                    types[7];
925     } RSA_Encrypt_COMMAND_DESCRIPTOR_t;
926     RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
927         /* entry         */     &TPM2_RSA_Encrypt,
928         /* inSize        */     (UINT16)(sizeof(RSA_Encrypt_In)),
929         /* outSize       */     (UINT16)(sizeof(RSA_Encrypt_Out)),
930         /* offsetOfTypes */     offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
931         /* offsets       */     {(UINT16)(offsetof(RSA_Encrypt_In, message)),
932                                   (UINT16)(offsetof(RSA_Encrypt_In, inScheme)),
933                                   (UINT16)(offsetof(RSA_Encrypt_In, label))},
934         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
935                                   TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
936                                   TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
937                                   TPM2B_DATA_P_UNMARSHAL,
938                                   END_OF_LIST,
939                                   TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
940                                   END_OF_LIST}
941     };
942     #define _RSA_EncryptDataAddress (&_RSA_EncryptData)
943     #else
944     #define _RSA_EncryptDataAddress 0
945     #endif // CC_RSA_Encrypt
946     #if CC_RSA_Decrypt
947     #include "RSA_Decrypt_fp.h"
948     typedef TPM_RC  (RSA_Decrypt_Entry)(
949         RSA_Decrypt_In              *in,
950         RSA_Decrypt_Out             *out
951     );
952     typedef const struct {
953         RSA_Decrypt_Entry       *entry;
954         UINT16                  inSize;
955         UINT16                  outSize;
956         UINT16                  offsetOfTypes;
957         UINT16                  paramOffsets[3];
```

```
958         BYTE                    types[7];
959     } RSA_Decrypt_COMMAND_DESCRIPTOR_t;
960     RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
961         /* entry          */    &TPM2_RSA_Decrypt,
962         /* inSize         */    (UINT16)(sizeof(RSA_Decrypt_In)),
963         /* outSize        */    (UINT16)(sizeof(RSA_Decrypt_Out)),
964         /* offsetOfTypes */    offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
965         /* offsets        */    {(UINT16)(offsetof(RSA_Decrypt_In, cipherText)),
966                                  (UINT16)(offsetof(RSA_Decrypt_In, inScheme)),
967                                  (UINT16)(offsetof(RSA_Decrypt_In, label))},
968         /* types          */    {TPMI_DH_OBJECT_H_UNMARSHAL,
969                                  TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
970                                  TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
971                                  TPM2B_DATA_P_UNMARSHAL,
972                                  END_OF_LIST,
973                                  TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
974                                  END_OF_LIST}
975     };
976     #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
977     #else
978     #define _RSA_DecryptDataAddress 0
979     #endif // CC_RSA_Decrypt
980     #if CC_ECDH_KeyGen
981     #include "ECDH_KeyGen_fp.h"
982     typedef TPM_RC  (ECDH_KeyGen_Entry)(
983         ECDH_KeyGen_In              *in,
984         ECDH_KeyGen_Out             *out
985     );
986     typedef const struct {
987         ECDH_KeyGen_Entry       *entry;
988         UINT16                  inSize;
989         UINT16                  outSize;
990         UINT16                  offsetOfTypes;
991         UINT16                  paramOffsets[1];
992         BYTE                    types[5];
993     } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
994     ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
995         /* entry          */    &TPM2_ECDH_KeyGen,
996         /* inSize         */    (UINT16)(sizeof(ECDH_KeyGen_In)),
997         /* outSize        */    (UINT16)(sizeof(ECDH_KeyGen_Out)),
998         /* offsetOfTypes */    offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
999         /* offsets        */    {(UINT16)(offsetof(ECDH_KeyGen_Out, pubPoint))},
1000        /* types          */    {TPMI_DH_OBJECT_H_UNMARSHAL,
1001                                 END_OF_LIST,
1002                                 TPM2B_ECC_POINT_P_MARSHAL,
1003                                 TPM2B_ECC_POINT_P_MARSHAL,
1004                                 END_OF_LIST}
1005    };
1006    #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
1007    #else
1008    #define _ECDH_KeyGenDataAddress 0
1009    #endif // CC_ECDH_KeyGen
1010    #if CC_ECDH_ZGen
1011    #include "ECDH_ZGen_fp.h"
1012    typedef TPM_RC  (ECDH_ZGen_Entry)(
1013        ECDH_ZGen_In                *in,
1014        ECDH_ZGen_Out               *out
1015    );
1016    typedef const struct {
1017        ECDH_ZGen_Entry         *entry;
1018        UINT16                  inSize;
1019        UINT16                  outSize;
1020        UINT16                  offsetOfTypes;
1021        UINT16                  paramOffsets[1];
1022        BYTE                    types[5];
1023    } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
```

```
1024    ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
1025        /* entry         */     &TPM2_ECDH_ZGen,
1026        /* inSize        */     (UINT16)(sizeof(ECDH_ZGen_In)),
1027        /* outSize       */     (UINT16)(sizeof(ECDH_ZGen_Out)),
1028        /* offsetOfTypes */     offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
1029        /* offsets       */     {(UINT16)(offsetof(ECDH_ZGen_In, inPoint))},
1030        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1031                                 TPM2B_ECC_POINT_P_UNMARSHAL,
1032                                 END_OF_LIST,
1033                                 TPM2B_ECC_POINT_P_MARSHAL,
1034                                 END_OF_LIST}
1035    };
1036    #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
1037    #else
1038    #define _ECDH_ZGenDataAddress 0
1039    #endif // CC_ECDH_ZGen
1040    #if CC_ECC_Parameters
1041    #include "ECC_Parameters_fp.h"
1042    typedef TPM_RC  (ECC_Parameters_Entry)(
1043        ECC_Parameters_In           *in,
1044        ECC_Parameters_Out          *out
1045    );
1046    typedef const struct {
1047        ECC_Parameters_Entry    *entry;
1048        UINT16                  inSize;
1049        UINT16                  outSize;
1050        UINT16                  offsetOfTypes;
1051        BYTE                    types[4];
1052    } ECC_Parameters_COMMAND_DESCRIPTOR_t;
1053    ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
1054        /* entry         */     &TPM2_ECC_Parameters,
1055        /* inSize        */     (UINT16)(sizeof(ECC_Parameters_In)),
1056        /* outSize       */     (UINT16)(sizeof(ECC_Parameters_Out)),
1057        /* offsetOfTypes */     offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
1058        /* offsets       */     // No parameter offsets;
1059        /* types         */     {TPMI_ECC_CURVE_P_UNMARSHAL,
1060                                 END_OF_LIST,
1061                                 TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1062                                 END_OF_LIST}
1063    };
1064    #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1065    #else
1066    #define _ECC_ParametersDataAddress 0
1067    #endif // CC_ECC_Parameters
1068    #if CC_ZGen_2Phase
1069    #include "ZGen_2Phase_fp.h"
1070    typedef TPM_RC  (ZGen_2Phase_Entry)(
1071        ZGen_2Phase_In              *in,
1072        ZGen_2Phase_Out             *out
1073    );
1074    typedef const struct {
1075        ZGen_2Phase_Entry       *entry;
1076        UINT16                  inSize;
1077        UINT16                  outSize;
1078        UINT16                  offsetOfTypes;
1079        UINT16                  paramOffsets[5];
1080        BYTE                    types[9];
1081    } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1082    ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
1083        /* entry         */     &TPM2_ZGen_2Phase,
1084        /* inSize        */     (UINT16)(sizeof(ZGen_2Phase_In)),
1085        /* outSize       */     (UINT16)(sizeof(ZGen_2Phase_Out)),
1086        /* offsetOfTypes */     offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1087        /* offsets       */     {(UINT16)(offsetof(ZGen_2Phase_In, inQsB)),
1088                                 (UINT16)(offsetof(ZGen_2Phase_In, inQeB)),
1089                                 (UINT16)(offsetof(ZGen_2Phase_In, inScheme)),
```

```
1090                                (UINT16)(offsetof(ZGen_2Phase_In, counter)),
1091                                (UINT16)(offsetof(ZGen_2Phase_Out, outZ2))},
1092        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1093                                TPM2B_ECC_POINT_P_UNMARSHAL,
1094                                TPM2B_ECC_POINT_P_UNMARSHAL,
1095                                TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1096                                UINT16_P_UNMARSHAL,
1097                                END_OF_LIST,
1098                                TPM2B_ECC_POINT_P_MARSHAL,
1099                                TPM2B_ECC_POINT_P_MARSHAL,
1100                                END_OF_LIST}
1101    };
1102    #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1103    #else
1104    #define _ZGen_2PhaseDataAddress 0
1105    #endif // CC_ZGen_2Phase
1106    #if CC_EncryptDecrypt
1107    #include "EncryptDecrypt_fp.h"
1108    typedef TPM_RC  (EncryptDecrypt_Entry)(
1109        EncryptDecrypt_In           *in,
1110        EncryptDecrypt_Out          *out
1111    );
1112    typedef const struct {
1113        EncryptDecrypt_Entry    *entry;
1114        UINT16                  inSize;
1115        UINT16                  outSize;
1116        UINT16                  offsetOfTypes;
1117        UINT16                  paramOffsets[5];
1118        BYTE                    types[9];
1119    } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1120    EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1121        /* entry         */     &TPM2_EncryptDecrypt,
1122        /* inSize        */     (UINT16)(sizeof(EncryptDecrypt_In)),
1123        /* outSize       */     (UINT16)(sizeof(EncryptDecrypt_Out)),
1124        /* offsetOfTypes */     offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1125        /* offsets       */     {(UINT16)(offsetof(EncryptDecrypt_In, decrypt)),
1126                                 (UINT16)(offsetof(EncryptDecrypt_In, mode)),
1127                                 (UINT16)(offsetof(EncryptDecrypt_In, ivIn)),
1128                                 (UINT16)(offsetof(EncryptDecrypt_In, inData)),
1129                                 (UINT16)(offsetof(EncryptDecrypt_Out, ivOut))},
1130        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1131                                TPMI_YES_NO_P_UNMARSHAL,
1132                                TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1133                                TPM2B_IV_P_UNMARSHAL,
1134                                TPM2B_MAX_BUFFER_P_UNMARSHAL,
1135                                END_OF_LIST,
1136                                TPM2B_MAX_BUFFER_P_MARSHAL,
1137                                TPM2B_IV_P_MARSHAL,
1138                                END_OF_LIST}
1139    };
1140    #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1141    #else
1142    #define _EncryptDecryptDataAddress 0
1143    #endif // CC_EncryptDecrypt
1144    #if CC_EncryptDecrypt2
1145    #include "EncryptDecrypt2_fp.h"
1146    typedef TPM_RC  (EncryptDecrypt2_Entry)(
1147        EncryptDecrypt2_In          *in,
1148        EncryptDecrypt2_Out         *out
1149    );
1150    typedef const struct {
1151        EncryptDecrypt2_Entry   *entry;
1152        UINT16                  inSize;
1153        UINT16                  outSize;
1154        UINT16                  offsetOfTypes;
1155        UINT16                  paramOffsets[5];
```

```
1156        BYTE                    types[9];
1157    } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1158    EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1159        /* entry          */    &TPM2_EncryptDecrypt2,
1160        /* inSize         */    (UINT16)(sizeof(EncryptDecrypt2_In)),
1161        /* outSize        */    (UINT16)(sizeof(EncryptDecrypt2_Out)),
1162        /* offsetOfTypes */    offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1163        /* offsets        */    {(UINT16)(offsetof(EncryptDecrypt2_In, inData)),
1164                                 (UINT16)(offsetof(EncryptDecrypt2_In, decrypt)),
1165                                 (UINT16)(offsetof(EncryptDecrypt2_In, mode)),
1166                                 (UINT16)(offsetof(EncryptDecrypt2_In, ivIn)),
1167                                 (UINT16)(offsetof(EncryptDecrypt2_Out, ivOut))},
1168        /* types          */    {TPMI_DH_OBJECT_H_UNMARSHAL,
1169                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1170                                 TPMI_YES_NO_P_UNMARSHAL,
1171                                 TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1172                                 TPM2B_IV_P_UNMARSHAL,
1173                                 END_OF_LIST,
1174                                 TPM2B_MAX_BUFFER_P_MARSHAL,
1175                                 TPM2B_IV_P_MARSHAL,
1176                                 END_OF_LIST}
1177    };
1178    #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1179    #else
1180    #define _EncryptDecrypt2DataAddress 0
1181    #endif // CC_EncryptDecrypt2
1182    #if CC_Hash
1183    #include "Hash_fp.h"
1184    typedef TPM_RC  (Hash_Entry)(
1185        Hash_In                     *in,
1186        Hash_Out                    *out
1187    );
1188    typedef const struct {
1189        Hash_Entry              *entry;
1190        UINT16                  inSize;
1191        UINT16                  outSize;
1192        UINT16                  offsetOfTypes;
1193        UINT16                  paramOffsets[3];
1194        BYTE                    types[7];
1195    } Hash_COMMAND_DESCRIPTOR_t;
1196    Hash_COMMAND_DESCRIPTOR_t _HashData = {
1197        /* entry          */    &TPM2_Hash,
1198        /* inSize         */    (UINT16)(sizeof(Hash_In)),
1199        /* outSize        */    (UINT16)(sizeof(Hash_Out)),
1200        /* offsetOfTypes */    offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1201        /* offsets        */    {(UINT16)(offsetof(Hash_In, hashAlg)),
1202                                 (UINT16)(offsetof(Hash_In, hierarchy)),
1203                                 (UINT16)(offsetof(Hash_Out, validation))},
1204        /* types          */    {TPM2B_MAX_BUFFER_P_UNMARSHAL,
1205                                 TPMI_ALG_HASH_P_UNMARSHAL,
1206                                 TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1207                                 END_OF_LIST,
1208                                 TPM2B_DIGEST_P_MARSHAL,
1209                                 TPMT_TK_HASHCHECK_P_MARSHAL,
1210                                 END_OF_LIST}
1211    };
1212    #define _HashDataAddress (&_HashData)
1213    #else
1214    #define _HashDataAddress 0
1215    #endif // CC_Hash
1216    #if CC_HMAC
1217    #include "HMAC_fp.h"
1218    typedef TPM_RC  (HMAC_Entry)(
1219        HMAC_In                     *in,
1220        HMAC_Out                    *out
1221    );
```

```
1222    typedef const struct {
1223        HMAC_Entry              *entry;
1224        UINT16                  inSize;
1225        UINT16                  outSize;
1226        UINT16                  offsetOfTypes;
1227        UINT16                  paramOffsets[2];
1228        BYTE                    types[6];
1229    } HMAC_COMMAND_DESCRIPTOR_t;
1230    HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1231        /* entry         */     &TPM2_HMAC,
1232        /* inSize        */     (UINT16)(sizeof(HMAC_In)),
1233        /* outSize       */     (UINT16)(sizeof(HMAC_Out)),
1234        /* offsetOfTypes */     offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1235        /* offsets       */     {(UINT16)(offsetof(HMAC_In, buffer)),
1236                                 (UINT16)(offsetof(HMAC_In, hashAlg))},
1237        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1238                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1239                                 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1240                                 END_OF_LIST,
1241                                 TPM2B_DIGEST_P_MARSHAL,
1242                                 END_OF_LIST}
1243    };
1244    #define _HMACDataAddress (&_HMACData)
1245    #else
1246    #define _HMACDataAddress 0
1247    #endif // CC_HMAC
1248    #if CC_MAC
1249    #include "MAC_fp.h"
1250    typedef TPM_RC  (MAC_Entry)(
1251        MAC_In                      *in,
1252        MAC_Out                     *out
1253    );
1254    typedef const struct {
1255        MAC_Entry               *entry;
1256        UINT16                  inSize;
1257        UINT16                  outSize;
1258        UINT16                  offsetOfTypes;
1259        UINT16                  paramOffsets[2];
1260        BYTE                    types[6];
1261    } MAC_COMMAND_DESCRIPTOR_t;
1262    MAC_COMMAND_DESCRIPTOR_t _MACData = {
1263        /* entry         */     &TPM2_MAC,
1264        /* inSize        */     (UINT16)(sizeof(MAC_In)),
1265        /* outSize       */     (UINT16)(sizeof(MAC_Out)),
1266        /* offsetOfTypes */     offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
1267        /* offsets       */     {(UINT16)(offsetof(MAC_In, buffer)),
1268                                 (UINT16)(offsetof(MAC_In, inScheme))},
1269        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1270                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1271                                 TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1272                                 END_OF_LIST,
1273                                 TPM2B_DIGEST_P_MARSHAL,
1274                                 END_OF_LIST}
1275    };
1276    #define _MACDataAddress (&_MACData)
1277    #else
1278    #define _MACDataAddress 0
1279    #endif // CC_MAC
1280    #if CC_GetRandom
1281    #include "GetRandom_fp.h"
1282    typedef TPM_RC  (GetRandom_Entry)(
1283        GetRandom_In                *in,
1284        GetRandom_Out               *out
1285    );
1286    typedef const struct {
1287        GetRandom_Entry         *entry;
```

```
1288        UINT16                  inSize;
1289        UINT16                  outSize;
1290        UINT16                  offsetOfTypes;
1291        BYTE                    types[4];
1292    } GetRandom_COMMAND_DESCRIPTOR_t;
1293    GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1294        /* entry         */     &TPM2_GetRandom,
1295        /* inSize        */     (UINT16)(sizeof(GetRandom_In)),
1296        /* outSize       */     (UINT16)(sizeof(GetRandom_Out)),
1297        /* offsetOfTypes */     offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
1298        /* offsets       */     // No parameter offsets;
1299        /* types         */     {UINT16_P_UNMARSHAL,
1300                                 END_OF_LIST,
1301                                 TPM2B_DIGEST_P_MARSHAL,
1302                                 END_OF_LIST}
1303    };
1304    #define _GetRandomDataAddress (&_GetRandomData)
1305    #else
1306    #define _GetRandomDataAddress 0
1307    #endif // CC_GetRandom
1308    #if CC_StirRandom
1309    #include "StirRandom_fp.h"
1310    typedef TPM_RC  (StirRandom_Entry)(
1311        StirRandom_In               *in
1312    );
1313    typedef const struct {
1314        StirRandom_Entry        *entry;
1315        UINT16                  inSize;
1316        UINT16                  outSize;
1317        UINT16                  offsetOfTypes;
1318        BYTE                    types[3];
1319    } StirRandom_COMMAND_DESCRIPTOR_t;
1320    StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
1321        /* entry         */     &TPM2_StirRandom,
1322        /* inSize        */     (UINT16)(sizeof(StirRandom_In)),
1323        /* outSize       */     0,
1324        /* offsetOfTypes */     offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1325        /* offsets       */     // No parameter offsets;
1326        /* types         */     {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1327                                 END_OF_LIST,
1328                                 END_OF_LIST}
1329    };
1330    #define _StirRandomDataAddress (&_StirRandomData)
1331    #else
1332    #define _StirRandomDataAddress 0
1333    #endif // CC_StirRandom
1334    #if CC_HMAC_Start
1335    #include "HMAC_Start_fp.h"
1336    typedef TPM_RC  (HMAC_Start_Entry)(
1337        HMAC_Start_In               *in,
1338        HMAC_Start_Out              *out
1339    );
1340    typedef const struct {
1341        HMAC_Start_Entry        *entry;
1342        UINT16                  inSize;
1343        UINT16                  outSize;
1344        UINT16                  offsetOfTypes;
1345        UINT16                  paramOffsets[2];
1346        BYTE                    types[6];
1347    } HMAC_Start_COMMAND_DESCRIPTOR_t;
1348    HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
1349        /* entry         */     &TPM2_HMAC_Start,
1350        /* inSize        */     (UINT16)(sizeof(HMAC_Start_In)),
1351        /* outSize       */     (UINT16)(sizeof(HMAC_Start_Out)),
1352        /* offsetOfTypes */     offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
1353        /* offsets       */     {(UINT16)(offsetof(HMAC_Start_In, auth)),
```

```
1354                                (UINT16)(offsetof(HMAC_Start_In, hashAlg))},
1355        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1356                                TPM2B_AUTH_P_UNMARSHAL,
1357                                TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1358                                END_OF_LIST,
1359                                TPMI_DH_OBJECT_H_MARSHAL,
1360                                END_OF_LIST}
1361    };
1362    #define _HMAC_StartDataAddress (&_HMAC_StartData)
1363    #else
1364    #define _HMAC_StartDataAddress 0
1365    #endif // CC_HMAC_Start
1366    #if CC_MAC_Start
1367    #include "MAC_Start_fp.h"
1368    typedef TPM_RC  (MAC_Start_Entry)(
1369        MAC_Start_In                *in,
1370        MAC_Start_Out               *out
1371    );
1372    typedef const struct {
1373        MAC_Start_Entry         *entry;
1374        UINT16                  inSize;
1375        UINT16                  outSize;
1376        UINT16                  offsetOfTypes;
1377        UINT16                  paramOffsets[2];
1378        BYTE                    types[6];
1379    } MAC_Start_COMMAND_DESCRIPTOR_t;
1380    MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
1381        /* entry         */     &TPM2_MAC_Start,
1382        /* inSize        */     (UINT16)(sizeof(MAC_Start_In)),
1383        /* outSize       */     (UINT16)(sizeof(MAC_Start_Out)),
1384        /* offsetOfTypes */     offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
1385        /* offsets       */     {(UINT16)(offsetof(MAC_Start_In, auth)),
1386                                 (UINT16)(offsetof(MAC_Start_In, inScheme))},
1387        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1388                                TPM2B_AUTH_P_UNMARSHAL,
1389                                TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1390                                END_OF_LIST,
1391                                TPMI_DH_OBJECT_H_MARSHAL,
1392                                END_OF_LIST}
1393    };
1394    #define _MAC_StartDataAddress (&_MAC_StartData)
1395    #else
1396    #define _MAC_StartDataAddress 0
1397    #endif // CC_MAC_Start
1398    #if CC_HashSequenceStart
1399    #include "HashSequenceStart_fp.h"
1400    typedef TPM_RC  (HashSequenceStart_Entry)(
1401        HashSequenceStart_In            *in,
1402        HashSequenceStart_Out           *out
1403    );
1404    typedef const struct {
1405        HashSequenceStart_Entry     *entry;
1406        UINT16                      inSize;
1407        UINT16                      outSize;
1408        UINT16                      offsetOfTypes;
1409        UINT16                      paramOffsets[1];
1410        BYTE                        types[5];
1411    } HashSequenceStart_COMMAND_DESCRIPTOR_t;
1412    HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
1413        /* entry         */         &TPM2_HashSequenceStart,
1414        /* inSize        */         (UINT16)(sizeof(HashSequenceStart_In)),
1415        /* outSize       */         (UINT16)(sizeof(HashSequenceStart_Out)),
1416        /* offsetOfTypes */         offsetof(HashSequenceStart_COMMAND_DESCRIPTOR_t,
    types),
1417        /* offsets       */         {(UINT16)(offsetof(HashSequenceStart_In, hashAlg))},
1418        /* types         */         {TPM2B_AUTH_P_UNMARSHAL,
```

```
                                    TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
                                    END_OF_LIST,
                                    TPMI_DH_OBJECT_H_MARSHAL,
                                    END_OF_LIST}
};
#define _HashSequenceStartDataAddress (&_HashSequenceStartData)
#else
#define _HashSequenceStartDataAddress 0
#endif // CC_HashSequenceStart
#if CC_SequenceUpdate
#include "SequenceUpdate_fp.h"
typedef TPM_RC  (SequenceUpdate_Entry)(
    SequenceUpdate_In           *in
);
typedef const struct {
    SequenceUpdate_Entry    *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[1];
    BYTE                    types[4];
} SequenceUpdate_COMMAND_DESCRIPTOR_t;
SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
    /* entry         */     &TPM2_SequenceUpdate,
    /* inSize        */     (UINT16)(sizeof(SequenceUpdate_In)),
    /* outSize       */     0,
    /* offsetOfTypes */     offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(SequenceUpdate_In, buffer))},
    /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
                             TPM2B_MAX_BUFFER_P_UNMARSHAL,
                             END_OF_LIST,
                             END_OF_LIST}
};
#define _SequenceUpdateDataAddress (&_SequenceUpdateData)
#else
#define _SequenceUpdateDataAddress 0
#endif // CC_SequenceUpdate
#if CC_SequenceComplete
#include "SequenceComplete_fp.h"
typedef TPM_RC  (SequenceComplete_Entry)(
    SequenceComplete_In         *in,
    SequenceComplete_Out        *out
);
typedef const struct {
    SequenceComplete_Entry  *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[3];
    BYTE                    types[7];
} SequenceComplete_COMMAND_DESCRIPTOR_t;
SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
    /* entry         */     &TPM2_SequenceComplete,
    /* inSize        */     (UINT16)(sizeof(SequenceComplete_In)),
    /* outSize       */     (UINT16)(sizeof(SequenceComplete_Out)),
    /* offsetOfTypes */     offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(SequenceComplete_In, buffer)),
                             (UINT16)(offsetof(SequenceComplete_In, hierarchy)),
                             (UINT16)(offsetof(SequenceComplete_Out, validation))},
    /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
                             TPM2B_MAX_BUFFER_P_UNMARSHAL,
                             TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
                             END_OF_LIST,
                             TPM2B_DIGEST_P_MARSHAL,
                             TPMT_TK_HASHCHECK_P_MARSHAL,
                             END_OF_LIST}
```

```
1485    };
1486    #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1487    #else
1488    #define _SequenceCompleteDataAddress 0
1489    #endif // CC_SequenceComplete
1490    #if CC_EventSequenceComplete
1491    #include "EventSequenceComplete_fp.h"
1492    typedef TPM_RC  (EventSequenceComplete_Entry)(
1493        EventSequenceComplete_In            *in,
1494        EventSequenceComplete_Out           *out
1495    );
1496    typedef const struct {
1497        EventSequenceComplete_Entry     *entry;
1498        UINT16                          inSize;
1499        UINT16                          outSize;
1500        UINT16                          offsetOfTypes;
1501        UINT16                          paramOffsets[2];
1502        BYTE                            types[6];
1503    } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1504    EventSequenceComplete_COMMAND_DESCRIPTOR_t _EventSequenceCompleteData = {
1505        /* entry         */             &TPM2_EventSequenceComplete,
1506        /* inSize        */             (UINT16)(sizeof(EventSequenceComplete_In)),
1507        /* outSize       */             (UINT16)(sizeof(EventSequenceComplete_Out)),
1508        /* offsetOfTypes */
        offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),
1509        /* offsets       */             {(UINT16)(offsetof(EventSequenceComplete_In,
        sequenceHandle)),
1510                                         (UINT16)(offsetof(EventSequenceComplete_In,
        buffer))},
1511        /* types         */             {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1512                                         TPMI_DH_OBJECT_H_UNMARSHAL,
1513                                         TPM2B_MAX_BUFFER_P_UNMARSHAL,
1514                                         END_OF_LIST,
1515                                         TPML_DIGEST_VALUES_P_MARSHAL,
1516                                         END_OF_LIST}
1517    };
1518    #define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
1519    #else
1520    #define _EventSequenceCompleteDataAddress 0
1521    #endif // CC_EventSequenceComplete
1522    #if CC_Certify
1523    #include "Certify_fp.h"
1524    typedef TPM_RC  (Certify_Entry)(
1525        Certify_In                  *in,
1526        Certify_Out                 *out
1527    );
1528    typedef const struct {
1529        Certify_Entry           *entry;
1530        UINT16                  inSize;
1531        UINT16                  outSize;
1532        UINT16                  offsetOfTypes;
1533        UINT16                  paramOffsets[4];
1534        BYTE                    types[8];
1535    } Certify_COMMAND_DESCRIPTOR_t;
1536    Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
1537        /* entry         */     &TPM2_Certify,
1538        /* inSize        */     (UINT16)(sizeof(Certify_In)),
1539        /* outSize       */     (UINT16)(sizeof(Certify_Out)),
1540        /* offsetOfTypes */     offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
1541        /* offsets       */     {(UINT16)(offsetof(Certify_In, signHandle)),
1542                                 (UINT16)(offsetof(Certify_In, qualifyingData)),
1543                                 (UINT16)(offsetof(Certify_In, inScheme)),
1544                                 (UINT16)(offsetof(Certify_Out, signature))},
1545        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1546                                 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1547                                 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)),
                             (UINT16)(offsetof(CertifyCreation_Out, signature))},
    /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
                             TPMI_DH_OBJECT_H_UNMARSHAL,
                             TPM2B_DATA_P_UNMARSHAL,
                             TPM2B_DIGEST_P_UNMARSHAL,
                             TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
                             TPMT_TK_CREATION_P_UNMARSHAL,
                             END_OF_LIST,
                             TPM2B_ATTEST_P_MARSHAL,
                             TPMT_SIGNATURE_P_MARSHAL,
                             END_OF_LIST}
};
#define _CertifyCreationDataAddress (&_CertifyCreationData)
#else
#define _CertifyCreationDataAddress 0
#endif // CC_CertifyCreation
#if CC_Quote
#include "Quote_fp.h"
typedef TPM_RC  (Quote_Entry)(
    Quote_In                    *in,
    Quote_Out                   *out
);
typedef const struct {
    Quote_Entry             *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[4];
    BYTE                    types[8];
} Quote_COMMAND_DESCRIPTOR_t;
Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
    /* entry         */     &TPM2_Quote,
```

```
1614        /* inSize         */     (UINT16)(sizeof(Quote_In)),
1615        /* outSize        */     (UINT16)(sizeof(Quote_Out)),
1616        /* offsetOfTypes */     offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
1617        /* offsets        */     {(UINT16)(offsetof(Quote_In, qualifyingData)),
1618                                  (UINT16)(offsetof(Quote_In, inScheme)),
1619                                  (UINT16)(offsetof(Quote_In, PCRselect)),
1620                                  (UINT16)(offsetof(Quote_Out, signature))},
1621        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1622                                  TPM2B_DATA_P_UNMARSHAL,
1623                                  TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1624                                  TPML_PCR_SELECTION_P_UNMARSHAL,
1625                                  END_OF_LIST,
1626                                  TPM2B_ATTEST_P_MARSHAL,
1627                                  TPMT_SIGNATURE_P_MARSHAL,
1628                                  END_OF_LIST}
1629    };
1630    #define _QuoteDataAddress (&_QuoteData)
1631    #else
1632    #define _QuoteDataAddress 0
1633    #endif // CC_Quote
1634    #if CC_GetSessionAuditDigest
1635    #include "GetSessionAuditDigest_fp.h"
1636    typedef TPM_RC  (GetSessionAuditDigest_Entry)(
1637        GetSessionAuditDigest_In            *in,
1638        GetSessionAuditDigest_Out           *out
1639    );
1640    typedef const struct {
1641        GetSessionAuditDigest_Entry     *entry;
1642        UINT16                          inSize;
1643        UINT16                          outSize;
1644        UINT16                          offsetOfTypes;
1645        UINT16                          paramOffsets[5];
1646        BYTE                            types[9];
1647    } GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
1648    GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
1649        /* entry          */            &TPM2_GetSessionAuditDigest,
1650        /* inSize         */            (UINT16)(sizeof(GetSessionAuditDigest_In)),
1651        /* outSize        */            (UINT16)(sizeof(GetSessionAuditDigest_Out)),
1652        /* offsetOfTypes */
    offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t, types),
1653        /* offsets        */            {(UINT16)(offsetof(GetSessionAuditDigest_In,
    signHandle)),
1654                                         (UINT16)(offsetof(GetSessionAuditDigest_In,
    sessionHandle)),
1655                                         (UINT16)(offsetof(GetSessionAuditDigest_In,
    qualifyingData)),
1656                                         (UINT16)(offsetof(GetSessionAuditDigest_In,
    inScheme)),
1657                                         (UINT16)(offsetof(GetSessionAuditDigest_Out,
    signature))},
1658        /* types          */            {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1659                                         TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1660                                         TPMI_SH_HMAC_H_UNMARSHAL,
1661                                         TPM2B_DATA_P_UNMARSHAL,
1662                                         TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1663                                         END_OF_LIST,
1664                                         TPM2B_ATTEST_P_MARSHAL,
1665                                         TPMT_SIGNATURE_P_MARSHAL,
1666                                         END_OF_LIST}
1667    };
1668    #define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
1669    #else
1670    #define _GetSessionAuditDigestDataAddress 0
1671    #endif // CC_GetSessionAuditDigest
1672    #if CC_GetCommandAuditDigest
1673    #include "GetCommandAuditDigest_fp.h"
```

```
1674    typedef TPM_RC  (GetCommandAuditDigest_Entry)(
1675        GetCommandAuditDigest_In            *in,
1676        GetCommandAuditDigest_Out           *out
1677    );
1678    typedef const struct {
1679        GetCommandAuditDigest_Entry     *entry;
1680        UINT16                          inSize;
1681        UINT16                          outSize;
1682        UINT16                          offsetOfTypes;
1683        UINT16                          paramOffsets[4];
1684        BYTE                            types[8];
1685    } GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
1686    GetCommandAuditDigest_COMMAND_DESCRIPTOR_t _GetCommandAuditDigestData = {
1687        /* entry         */             &TPM2_GetCommandAuditDigest,
1688        /* inSize        */             (UINT16)(sizeof(GetCommandAuditDigest_In)),
1689        /* outSize       */             (UINT16)(sizeof(GetCommandAuditDigest_Out)),
1690        /* offsetOfTypes */
        offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t, types),
1691        /* offsets       */             {(UINT16)(offsetof(GetCommandAuditDigest_In,
        signHandle)),
1692                                         (UINT16)(offsetof(GetCommandAuditDigest_In,
        qualifyingData)),
1693                                         (UINT16)(offsetof(GetCommandAuditDigest_In,
        inScheme)),
1694                                         (UINT16)(offsetof(GetCommandAuditDigest_Out,
        signature))},
1695        /* types         */             {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1696                                         TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1697                                         TPM2B_DATA_P_UNMARSHAL,
1698                                         TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1699                                         END_OF_LIST,
1700                                         TPM2B_ATTEST_P_MARSHAL,
1701                                         TPMT_SIGNATURE_P_MARSHAL,
1702                                         END_OF_LIST}
1703    };
1704    #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
1705    #else
1706    #define _GetCommandAuditDigestDataAddress 0
1707    #endif // CC_GetCommandAuditDigest
1708    #if CC_GetTime
1709    #include "GetTime_fp.h"
1710    typedef TPM_RC  (GetTime_Entry)(
1711        GetTime_In                  *in,
1712        GetTime_Out                 *out
1713    );
1714    typedef const struct {
1715        GetTime_Entry           *entry;
1716        UINT16                  inSize;
1717        UINT16                  outSize;
1718        UINT16                  offsetOfTypes;
1719        UINT16                  paramOffsets[4];
1720        BYTE                    types[8];
1721    } GetTime_COMMAND_DESCRIPTOR_t;
1722    GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
1723        /* entry         */     &TPM2_GetTime,
1724        /* inSize        */     (UINT16)(sizeof(GetTime_In)),
1725        /* outSize       */     (UINT16)(sizeof(GetTime_Out)),
1726        /* offsetOfTypes */     offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
1727        /* offsets       */     {(UINT16)(offsetof(GetTime_In, signHandle)),
1728                                 (UINT16)(offsetof(GetTime_In, qualifyingData)),
1729                                 (UINT16)(offsetof(GetTime_In, inScheme)),
1730                                 (UINT16)(offsetof(GetTime_Out, signature))},
1731        /* types         */     {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1732                                 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1733                                 TPM2B_DATA_P_UNMARSHAL,
1734                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
```

```
                                END_OF_LIST,
                                TPM2B_ATTEST_P_MARSHAL,
                                TPMT_SIGNATURE_P_MARSHAL,
                                END_OF_LIST}
};
#define _GetTimeDataAddress (&_GetTimeData)
#else
#define _GetTimeDataAddress 0
#endif // CC_GetTime
#if CC_CertifyX509
#include "CertifyX509_fp.h"
typedef TPM_RC  (CertifyX509_Entry)(
    CertifyX509_In              *in,
    CertifyX509_Out             *out
);
typedef const struct {
    CertifyX509_Entry       *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[6];
    BYTE                    types[10];
} CertifyX509_COMMAND_DESCRIPTOR_t;
CertifyX509_COMMAND_DESCRIPTOR_t _CertifyX509Data = {
    /* entry         */     &TPM2_CertifyX509,
    /* inSize        */     (UINT16)(sizeof(CertifyX509_In)),
    /* outSize       */     (UINT16)(sizeof(CertifyX509_Out)),
    /* offsetOfTypes */     offsetof(CertifyX509_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(CertifyX509_In, signHandle)),
                             (UINT16)(offsetof(CertifyX509_In, reserved)),
                             (UINT16)(offsetof(CertifyX509_In, inScheme)),
                             (UINT16)(offsetof(CertifyX509_In, partialCertificate)),
                             (UINT16)(offsetof(CertifyX509_Out, tbsDigest)),
                             (UINT16)(offsetof(CertifyX509_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,
                             TPM2B_MAX_BUFFER_P_UNMARSHAL,
                             END_OF_LIST,
                             TPM2B_MAX_BUFFER_P_MARSHAL,
                             TPM2B_DIGEST_P_MARSHAL,
                             TPMT_SIGNATURE_P_MARSHAL,
                             END_OF_LIST}
};
#define _CertifyX509DataAddress (&_CertifyX509Data)
#else
#define _CertifyX509DataAddress 0
#endif // CC_CertifyX509
#if CC_Commit
#include "Commit_fp.h"
typedef TPM_RC  (Commit_Entry)(
    Commit_In                   *in,
    Commit_Out                  *out
);
typedef const struct {
    Commit_Entry            *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[6];
    BYTE                    types[10];
} Commit_COMMAND_DESCRIPTOR_t;
Commit_COMMAND_DESCRIPTOR_t _CommitData = {
    /* entry         */     &TPM2_Commit,
    /* inSize        */     (UINT16)(sizeof(Commit_In)),
```

```
1801        /* outSize        */     (UINT16)(sizeof(Commit_Out)),
1802        /* offsetOfTypes  */     offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
1803        /* offsets        */     {(UINT16)(offsetof(Commit_In, P1)),
1804                                   (UINT16)(offsetof(Commit_In, s2)),
1805                                   (UINT16)(offsetof(Commit_In, y2)),
1806                                   (UINT16)(offsetof(Commit_Out, L)),
1807                                   (UINT16)(offsetof(Commit_Out, E)),
1808                                   (UINT16)(offsetof(Commit_Out, counter))},
1809        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1810                                   TPM2B_ECC_POINT_P_UNMARSHAL,
1811                                   TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1812                                   TPM2B_ECC_PARAMETER_P_UNMARSHAL,
1813                                   END_OF_LIST,
1814                                   TPM2B_ECC_POINT_P_MARSHAL,
1815                                   TPM2B_ECC_POINT_P_MARSHAL,
1816                                   TPM2B_ECC_POINT_P_MARSHAL,
1817                                   UINT16_P_MARSHAL,
1818                                   END_OF_LIST}
1819    };
1820    #define _CommitDataAddress (&_CommitData)
1821    #else
1822    #define _CommitDataAddress 0
1823    #endif // CC_Commit
1824    #if CC_EC_Ephemeral
1825    #include "EC_Ephemeral_fp.h"
1826    typedef TPM_RC  (EC_Ephemeral_Entry)(
1827        EC_Ephemeral_In             *in,
1828        EC_Ephemeral_Out            *out
1829    );
1830    typedef const struct {
1831        EC_Ephemeral_Entry      *entry;
1832        UINT16                  inSize;
1833        UINT16                  outSize;
1834        UINT16                  offsetOfTypes;
1835        UINT16                  paramOffsets[1];
1836        BYTE                    types[5];
1837    } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
1838    EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
1839        /* entry          */     &TPM2_EC_Ephemeral,
1840        /* inSize         */     (UINT16)(sizeof(EC_Ephemeral_In)),
1841        /* outSize        */     (UINT16)(sizeof(EC_Ephemeral_Out)),
1842        /* offsetOfTypes  */     offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
1843        /* offsets        */     {(UINT16)(offsetof(EC_Ephemeral_Out, counter))},
1844        /* types          */     {TPMI_ECC_CURVE_P_UNMARSHAL,
1845                                   END_OF_LIST,
1846                                   TPM2B_ECC_POINT_P_MARSHAL,
1847                                   UINT16_P_MARSHAL,
1848                                   END_OF_LIST}
1849    };
1850    #define _EC_EphemeralDataAddress (&_EC_EphemeralData)
1851    #else
1852    #define _EC_EphemeralDataAddress 0
1853    #endif // CC_EC_Ephemeral
1854    #if CC_VerifySignature
1855    #include "VerifySignature_fp.h"
1856    typedef TPM_RC  (VerifySignature_Entry)(
1857        VerifySignature_In          *in,
1858        VerifySignature_Out         *out
1859    );
1860    typedef const struct {
1861        VerifySignature_Entry   *entry;
1862        UINT16                  inSize;
1863        UINT16                  outSize;
1864        UINT16                  offsetOfTypes;
1865        UINT16                  paramOffsets[2];
1866        BYTE                    types[6];
```

```
1867  } VerifySignature_COMMAND_DESCRIPTOR_t;
1868  VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
1869      /* entry          */     &TPM2_VerifySignature,
1870      /* inSize         */     (UINT16)(sizeof(VerifySignature_In)),
1871      /* outSize        */     (UINT16)(sizeof(VerifySignature_Out)),
1872      /* offsetOfTypes */     offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
1873      /* offsets        */     {(UINT16)(offsetof(VerifySignature_In, digest)),
1874                                (UINT16)(offsetof(VerifySignature_In, signature))},
1875      /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1876                                TPM2B_DIGEST_P_UNMARSHAL,
1877                                TPMT_SIGNATURE_P_UNMARSHAL,
1878                                END_OF_LIST,
1879                                TPMT_TK_VERIFIED_P_MARSHAL,
1880                                END_OF_LIST}
1881  };
1882  #define _VerifySignatureDataAddress (&_VerifySignatureData)
1883  #else
1884  #define _VerifySignatureDataAddress 0
1885  #endif // CC_VerifySignature
1886  #if CC_Sign
1887  #include "Sign_fp.h"
1888  typedef TPM_RC  (Sign_Entry)(
1889      Sign_In                     *in,
1890      Sign_Out                    *out
1891  );
1892  typedef const struct {
1893      Sign_Entry              *entry;
1894      UINT16                  inSize;
1895      UINT16                  outSize;
1896      UINT16                  offsetOfTypes;
1897      UINT16                  paramOffsets[3];
1898      BYTE                    types[7];
1899  } Sign_COMMAND_DESCRIPTOR_t;
1900  Sign_COMMAND_DESCRIPTOR_t _SignData = {
1901      /* entry          */     &TPM2_Sign,
1902      /* inSize         */     (UINT16)(sizeof(Sign_In)),
1903      /* outSize        */     (UINT16)(sizeof(Sign_Out)),
1904      /* offsetOfTypes */     offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
1905      /* offsets        */     {(UINT16)(offsetof(Sign_In, digest)),
1906                                (UINT16)(offsetof(Sign_In, inScheme)),
1907                                (UINT16)(offsetof(Sign_In, validation))},
1908      /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1909                                TPM2B_DIGEST_P_UNMARSHAL,
1910                                TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1911                                TPMT_TK_HASHCHECK_P_UNMARSHAL,
1912                                END_OF_LIST,
1913                                TPMT_SIGNATURE_P_MARSHAL,
1914                                END_OF_LIST}
1915  };
1916  #define _SignDataAddress (&_SignData)
1917  #else
1918  #define _SignDataAddress 0
1919  #endif // CC_Sign
1920  #if CC_SetCommandCodeAuditStatus
1921  #include "SetCommandCodeAuditStatus_fp.h"
1922  typedef TPM_RC  (SetCommandCodeAuditStatus_Entry)(
1923      SetCommandCodeAuditStatus_In            *in
1924  );
1925  typedef const struct {
1926      SetCommandCodeAuditStatus_Entry     *entry;
1927      UINT16                              inSize;
1928      UINT16                              outSize;
1929      UINT16                              offsetOfTypes;
1930      UINT16                              paramOffsets[3];
1931      BYTE                                types[6];
1932  } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
```

```
1933    SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
1934        /* entry         */                &TPM2_SetCommandCodeAuditStatus,
1935        /* inSize        */
        (UINT16)(sizeof(SetCommandCodeAuditStatus_In)),
1936        /* outSize       */                0,
1937        /* offsetOfTypes */
        offsetof(SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
1938        /* offsets       */
        {(UINT16)(offsetof(SetCommandCodeAuditStatus_In, auditAlg)),
1939
        (UINT16)(offsetof(SetCommandCodeAuditStatus_In, setList)),
1940
        (UINT16)(offsetof(SetCommandCodeAuditStatus_In, clearList))},
1941        /* types         */                {TPMI_RH_PROVISION_H_UNMARSHAL,
1942                                           TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1943                                           TPML_CC_P_UNMARSHAL,
1944                                           TPML_CC_P_UNMARSHAL,
1945                                           END_OF_LIST,
1946                                           END_OF_LIST}
1947    };
1948    #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
1949    #else
1950    #define _SetCommandCodeAuditStatusDataAddress 0
1951    #endif // CC_SetCommandCodeAuditStatus
1952    #if CC_PCR_Extend
1953    #include "PCR_Extend_fp.h"
1954    typedef TPM_RC  (PCR_Extend_Entry)(
1955        PCR_Extend_In               *in
1956    );
1957    typedef const struct {
1958        PCR_Extend_Entry        *entry;
1959        UINT16                  inSize;
1960        UINT16                  outSize;
1961        UINT16                  offsetOfTypes;
1962        UINT16                  paramOffsets[1];
1963        BYTE                    types[4];
1964    } PCR_Extend_COMMAND_DESCRIPTOR_t;
1965    PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
1966        /* entry         */     &TPM2_PCR_Extend,
1967        /* inSize        */     (UINT16)(sizeof(PCR_Extend_In)),
1968        /* outSize       */     0,
1969        /* offsetOfTypes */     offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
1970        /* offsets       */     {(UINT16)(offsetof(PCR_Extend_In, digests))},
1971        /* types         */     {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1972                                 TPML_DIGEST_VALUES_P_UNMARSHAL,
1973                                 END_OF_LIST,
1974                                 END_OF_LIST}
1975    };
1976    #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
1977    #else
1978    #define _PCR_ExtendDataAddress 0
1979    #endif // CC_PCR_Extend
1980    #if CC_PCR_Event
1981    #include "PCR_Event_fp.h"
1982    typedef TPM_RC  (PCR_Event_Entry)(
1983        PCR_Event_In                *in,
1984        PCR_Event_Out               *out
1985    );
1986    typedef const struct {
1987        PCR_Event_Entry         *entry;
1988        UINT16                  inSize;
1989        UINT16                  outSize;
1990        UINT16                  offsetOfTypes;
1991        UINT16                  paramOffsets[1];
1992        BYTE                    types[5];
1993    } PCR_Event_COMMAND_DESCRIPTOR_t;
```

```
1994  PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
1995      /* entry         */     &TPM2_PCR_Event,
1996      /* inSize        */     (UINT16)(sizeof(PCR_Event_In)),
1997      /* outSize       */     (UINT16)(sizeof(PCR_Event_Out)),
1998      /* offsetOfTypes */     offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
1999      /* offsets       */     {(UINT16)(offsetof(PCR_Event_In, eventData))},
2000      /* types         */     {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
2001                               TPM2B_EVENT_P_UNMARSHAL,
2002                               END_OF_LIST,
2003                               TPML_DIGEST_VALUES_P_MARSHAL,
2004                               END_OF_LIST}
2005  };
2006  #define _PCR_EventDataAddress (&_PCR_EventData)
2007  #else
2008  #define _PCR_EventDataAddress 0
2009  #endif // CC_PCR_Event
2010  #if CC_PCR_Read
2011  #include "PCR_Read_fp.h"
2012  typedef TPM_RC  (PCR_Read_Entry)(
2013      PCR_Read_In                 *in,
2014      PCR_Read_Out                *out
2015  );
2016  typedef const struct {
2017      PCR_Read_Entry          *entry;
2018      UINT16                  inSize;
2019      UINT16                  outSize;
2020      UINT16                  offsetOfTypes;
2021      UINT16                  paramOffsets[2];
2022      BYTE                    types[6];
2023  } PCR_Read_COMMAND_DESCRIPTOR_t;
2024  PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
2025      /* entry         */     &TPM2_PCR_Read,
2026      /* inSize        */     (UINT16)(sizeof(PCR_Read_In)),
2027      /* outSize       */     (UINT16)(sizeof(PCR_Read_Out)),
2028      /* offsetOfTypes */     offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
2029      /* offsets       */     {(UINT16)(offsetof(PCR_Read_Out, pcrSelectionOut)),
2030                                (UINT16)(offsetof(PCR_Read_Out, pcrValues))},
2031      /* types         */     {TPML_PCR_SELECTION_P_UNMARSHAL,
2032                               END_OF_LIST,
2033                               UINT32_P_MARSHAL,
2034                               TPML_PCR_SELECTION_P_MARSHAL,
2035                               TPML_DIGEST_P_MARSHAL,
2036                               END_OF_LIST}
2037  };
2038  #define _PCR_ReadDataAddress (&_PCR_ReadData)
2039  #else
2040  #define _PCR_ReadDataAddress 0
2041  #endif // CC_PCR_Read
2042  #if CC_PCR_Allocate
2043  #include "PCR_Allocate_fp.h"
2044  typedef TPM_RC  (PCR_Allocate_Entry)(
2045      PCR_Allocate_In             *in,
2046      PCR_Allocate_Out            *out
2047  );
2048  typedef const struct {
2049      PCR_Allocate_Entry      *entry;
2050      UINT16                  inSize;
2051      UINT16                  outSize;
2052      UINT16                  offsetOfTypes;
2053      UINT16                  paramOffsets[4];
2054      BYTE                    types[8];
2055  } PCR_Allocate_COMMAND_DESCRIPTOR_t;
2056  PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
2057      /* entry         */     &TPM2_PCR_Allocate,
2058      /* inSize        */     (UINT16)(sizeof(PCR_Allocate_In)),
2059      /* outSize       */     (UINT16)(sizeof(PCR_Allocate_Out)),
```

```
2060        /* offsetOfTypes */     offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
2061        /* offsets       */     {(UINT16)(offsetof(PCR_Allocate_In, pcrAllocation)),
2062                                  (UINT16)(offsetof(PCR_Allocate_Out, maxPCR)),
2063                                  (UINT16)(offsetof(PCR_Allocate_Out, sizeNeeded)),
2064                                  (UINT16)(offsetof(PCR_Allocate_Out, sizeAvailable))},
2065        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
2066                                  TPML_PCR_SELECTION_P_UNMARSHAL,
2067                                  END_OF_LIST,
2068                                  TPMI_YES_NO_P_MARSHAL,
2069                                  UINT32_P_MARSHAL,
2070                                  UINT32_P_MARSHAL,
2071                                  UINT32_P_MARSHAL,
2072                                  END_OF_LIST}
2073    };
2074    #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
2075    #else
2076    #define _PCR_AllocateDataAddress 0
2077    #endif // CC_PCR_Allocate
2078    #if CC_PCR_SetAuthPolicy
2079    #include "PCR_SetAuthPolicy_fp.h"
2080    typedef TPM_RC  (PCR_SetAuthPolicy_Entry)(
2081        PCR_SetAuthPolicy_In            *in
2082    );
2083    typedef const struct {
2084        PCR_SetAuthPolicy_Entry     *entry;
2085        UINT16                      inSize;
2086        UINT16                      outSize;
2087        UINT16                      offsetOfTypes;
2088        UINT16                      paramOffsets[3];
2089        BYTE                        types[6];
2090    } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
2091    PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t _PCR_SetAuthPolicyData = {
2092        /* entry         */         &TPM2_PCR_SetAuthPolicy,
2093        /* inSize        */         (UINT16)(sizeof(PCR_SetAuthPolicy_In)),
2094        /* outSize       */         0,
2095        /* offsetOfTypes */         offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
    types),
2096        /* offsets       */         {(UINT16)(offsetof(PCR_SetAuthPolicy_In, authPolicy)),
2097                                     (UINT16)(offsetof(PCR_SetAuthPolicy_In, hashAlg)),
2098                                     (UINT16)(offsetof(PCR_SetAuthPolicy_In, pcrNum))},
2099        /* types         */         {TPMI_RH_PLATFORM_H_UNMARSHAL,
2100                                     TPM2B_DIGEST_P_UNMARSHAL,
2101                                     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2102                                     TPMI_DH_PCR_P_UNMARSHAL,
2103                                     END_OF_LIST,
2104                                     END_OF_LIST}
2105    };
2106    #define _PCR_SetAuthPolicyDataAddress (&_PCR_SetAuthPolicyData)
2107    #else
2108    #define _PCR_SetAuthPolicyDataAddress 0
2109    #endif // CC_PCR_SetAuthPolicy
2110    #if CC_PCR_SetAuthValue
2111    #include "PCR_SetAuthValue_fp.h"
2112    typedef TPM_RC  (PCR_SetAuthValue_Entry)(
2113        PCR_SetAuthValue_In         *in
2114    );
2115    typedef const struct {
2116        PCR_SetAuthValue_Entry  *entry;
2117        UINT16                  inSize;
2118        UINT16                  outSize;
2119        UINT16                  offsetOfTypes;
2120        UINT16                  paramOffsets[1];
2121        BYTE                    types[4];
2122    } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
2123    PCR_SetAuthValue_COMMAND_DESCRIPTOR_t _PCR_SetAuthValueData = {
2124        /* entry         */     &TPM2_PCR_SetAuthValue,
```

```
2125        /* inSize         */     (UINT16)(sizeof(PCR_SetAuthValue_In)),
2126        /* outSize        */     0,
2127        /* offsetOfTypes  */     offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
2128        /* offsets        */     {(UINT16)(offsetof(PCR_SetAuthValue_In, auth))},
2129        /* types          */     {TPMI_DH_PCR_H_UNMARSHAL,
2130                                   TPM2B_DIGEST_P_UNMARSHAL,
2131                                   END_OF_LIST,
2132                                   END_OF_LIST}
2133    };
2134    #define _PCR_SetAuthValueDataAddress (&_PCR_SetAuthValueData)
2135    #else
2136    #define _PCR_SetAuthValueDataAddress 0
2137    #endif // CC_PCR_SetAuthValue
2138    #if CC_PCR_Reset
2139    #include "PCR_Reset_fp.h"
2140    typedef TPM_RC  (PCR_Reset_Entry)(
2141        PCR_Reset_In                *in
2142    );
2143    typedef const struct {
2144        PCR_Reset_Entry         *entry;
2145        UINT16                  inSize;
2146        UINT16                  outSize;
2147        UINT16                  offsetOfTypes;
2148        BYTE                    types[3];
2149    } PCR_Reset_COMMAND_DESCRIPTOR_t;
2150    PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2151        /* entry          */     &TPM2_PCR_Reset,
2152        /* inSize         */     (UINT16)(sizeof(PCR_Reset_In)),
2153        /* outSize        */     0,
2154        /* offsetOfTypes  */     offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2155        /* offsets        */     // No parameter offsets;
2156        /* types          */     {TPMI_DH_PCR_H_UNMARSHAL,
2157                                   END_OF_LIST,
2158                                   END_OF_LIST}
2159    };
2160    #define _PCR_ResetDataAddress (&_PCR_ResetData)
2161    #else
2162    #define _PCR_ResetDataAddress 0
2163    #endif // CC_PCR_Reset
2164    #if CC_PolicySigned
2165    #include "PolicySigned_fp.h"
2166    typedef TPM_RC  (PolicySigned_Entry)(
2167        PolicySigned_In             *in,
2168        PolicySigned_Out            *out
2169    );
2170    typedef const struct {
2171        PolicySigned_Entry      *entry;
2172        UINT16                  inSize;
2173        UINT16                  outSize;
2174        UINT16                  offsetOfTypes;
2175        UINT16                  paramOffsets[7];
2176        BYTE                    types[11];
2177    } PolicySigned_COMMAND_DESCRIPTOR_t;
2178    PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2179        /* entry          */     &TPM2_PolicySigned,
2180        /* inSize         */     (UINT16)(sizeof(PolicySigned_In)),
2181        /* outSize        */     (UINT16)(sizeof(PolicySigned_Out)),
2182        /* offsetOfTypes  */     offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2183        /* offsets        */     {(UINT16)(offsetof(PolicySigned_In, policySession)),
2184                                   (UINT16)(offsetof(PolicySigned_In, nonceTPM)),
2185                                   (UINT16)(offsetof(PolicySigned_In, cpHashA)),
2186                                   (UINT16)(offsetof(PolicySigned_In, policyRef)),
2187                                   (UINT16)(offsetof(PolicySigned_In, expiration)),
2188                                   (UINT16)(offsetof(PolicySigned_In, auth)),
2189                                   (UINT16)(offsetof(PolicySigned_Out, policyTicket))},
2190        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
```

```
                                TPMI_SH_POLICY_H_UNMARSHAL,
                                TPM2B_NONCE_P_UNMARSHAL,
                                TPM2B_DIGEST_P_UNMARSHAL,
                                TPM2B_NONCE_P_UNMARSHAL,
                                INT32_P_UNMARSHAL,
                                TPMT_SIGNATURE_P_UNMARSHAL,
                                END_OF_LIST,
                                TPM2B_TIMEOUT_P_MARSHAL,
                                TPMT_TK_AUTH_P_MARSHAL,
                                END_OF_LIST}
};
#define _PolicySignedDataAddress (&_PolicySignedData)
#else
#define _PolicySignedDataAddress 0
#endif // CC_PolicySigned
#if CC_PolicySecret
#include "PolicySecret_fp.h"
typedef TPM_RC  (PolicySecret_Entry)(
    PolicySecret_In             *in,
    PolicySecret_Out            *out
);
typedef const struct {
    PolicySecret_Entry      *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[6];
    BYTE                    types[10];
} PolicySecret_COMMAND_DESCRIPTOR_t;
PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
    /* entry         */     &TPM2_PolicySecret,
    /* inSize        */     (UINT16)(sizeof(PolicySecret_In)),
    /* outSize       */     (UINT16)(sizeof(PolicySecret_Out)),
    /* offsetOfTypes */     offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(PolicySecret_In, policySession)),
                             (UINT16)(offsetof(PolicySecret_In, nonceTPM)),
                             (UINT16)(offsetof(PolicySecret_In, cpHashA)),
                             (UINT16)(offsetof(PolicySecret_In, policyRef)),
                             (UINT16)(offsetof(PolicySecret_In, expiration)),
                             (UINT16)(offsetof(PolicySecret_Out, policyTicket))},
    /* types         */     {TPMI_DH_ENTITY_H_UNMARSHAL,
                             TPMI_SH_POLICY_H_UNMARSHAL,
                             TPM2B_NONCE_P_UNMARSHAL,
                             TPM2B_DIGEST_P_UNMARSHAL,
                             TPM2B_NONCE_P_UNMARSHAL,
                             INT32_P_UNMARSHAL,
                             END_OF_LIST,
                             TPM2B_TIMEOUT_P_MARSHAL,
                             TPMT_TK_AUTH_P_MARSHAL,
                             END_OF_LIST}
};
#define _PolicySecretDataAddress (&_PolicySecretData)
#else
#define _PolicySecretDataAddress 0
#endif // CC_PolicySecret
#if CC_PolicyTicket
#include "PolicyTicket_fp.h"
typedef TPM_RC  (PolicyTicket_Entry)(
    PolicyTicket_In             *in
);
typedef const struct {
    PolicyTicket_Entry      *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[5];
```

```
2257        BYTE                    types[8];
2258    } PolicyTicket_COMMAND_DESCRIPTOR_t;
2259    PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2260        /* entry         */     &TPM2_PolicyTicket,
2261        /* inSize        */     (UINT16)(sizeof(PolicyTicket_In)),
2262        /* outSize       */     0,
2263        /* offsetOfTypes */     offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
2264        /* offsets       */     {(UINT16)(offsetof(PolicyTicket_In, timeout)),
2265                                 (UINT16)(offsetof(PolicyTicket_In, cpHashA)),
2266                                 (UINT16)(offsetof(PolicyTicket_In, policyRef)),
2267                                 (UINT16)(offsetof(PolicyTicket_In, authName)),
2268                                 (UINT16)(offsetof(PolicyTicket_In, ticket))},
2269        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2270                                 TPM2B_TIMEOUT_P_UNMARSHAL,
2271                                 TPM2B_DIGEST_P_UNMARSHAL,
2272                                 TPM2B_NONCE_P_UNMARSHAL,
2273                                 TPM2B_NAME_P_UNMARSHAL,
2274                                 TPMT_TK_AUTH_P_UNMARSHAL,
2275                                 END_OF_LIST,
2276                                 END_OF_LIST}
2277    };
2278    #define _PolicyTicketDataAddress (&_PolicyTicketData)
2279    #else
2280    #define _PolicyTicketDataAddress 0
2281    #endif // CC_PolicyTicket
2282    #if CC_PolicyOR
2283    #include "PolicyOR_fp.h"
2284    typedef TPM_RC  (PolicyOR_Entry)(
2285        PolicyOR_In                 *in
2286    );
2287    typedef const struct {
2288        PolicyOR_Entry          *entry;
2289        UINT16                  inSize;
2290        UINT16                  outSize;
2291        UINT16                  offsetOfTypes;
2292        UINT16                  paramOffsets[1];
2293        BYTE                    types[4];
2294    } PolicyOR_COMMAND_DESCRIPTOR_t;
2295    PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2296        /* entry         */     &TPM2_PolicyOR,
2297        /* inSize        */     (UINT16)(sizeof(PolicyOR_In)),
2298        /* outSize       */     0,
2299        /* offsetOfTypes */     offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2300        /* offsets       */     {(UINT16)(offsetof(PolicyOR_In, pHashList))},
2301        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2302                                 TPML_DIGEST_P_UNMARSHAL,
2303                                 END_OF_LIST,
2304                                 END_OF_LIST}
2305    };
2306    #define _PolicyORDataAddress (&_PolicyORData)
2307    #else
2308    #define _PolicyORDataAddress 0
2309    #endif // CC_PolicyOR
2310    #if CC_PolicyPCR
2311    #include "PolicyPCR_fp.h"
2312    typedef TPM_RC  (PolicyPCR_Entry)(
2313        PolicyPCR_In                *in
2314    );
2315    typedef const struct {
2316        PolicyPCR_Entry         *entry;
2317        UINT16                  inSize;
2318        UINT16                  outSize;
2319        UINT16                  offsetOfTypes;
2320        UINT16                  paramOffsets[2];
2321        BYTE                    types[5];
2322    } PolicyPCR_COMMAND_DESCRIPTOR_t;
```

```
2323    PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
2324        /* entry         */     &TPM2_PolicyPCR,
2325        /* inSize        */     (UINT16)(sizeof(PolicyPCR_In)),
2326        /* outSize       */     0,
2327        /* offsetOfTypes */     offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2328        /* offsets       */     {(UINT16)(offsetof(PolicyPCR_In, pcrDigest)),
2329                                 (UINT16)(offsetof(PolicyPCR_In, pcrs))},
2330        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2331                                 TPM2B_DIGEST_P_UNMARSHAL,
2332                                 TPML_PCR_SELECTION_P_UNMARSHAL,
2333                                 END_OF_LIST,
2334                                 END_OF_LIST}
2335    };
2336    #define _PolicyPCRDataAddress (&_PolicyPCRData)
2337    #else
2338    #define _PolicyPCRDataAddress 0
2339    #endif // CC_PolicyPCR
2340    #if CC_PolicyLocality
2341    #include "PolicyLocality_fp.h"
2342    typedef TPM_RC  (PolicyLocality_Entry)(
2343        PolicyLocality_In           *in
2344    );
2345    typedef const struct {
2346        PolicyLocality_Entry    *entry;
2347        UINT16                  inSize;
2348        UINT16                  outSize;
2349        UINT16                  offsetOfTypes;
2350        UINT16                  paramOffsets[1];
2351        BYTE                    types[4];
2352    } PolicyLocality_COMMAND_DESCRIPTOR_t;
2353    PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2354        /* entry         */     &TPM2_PolicyLocality,
2355        /* inSize        */     (UINT16)(sizeof(PolicyLocality_In)),
2356        /* outSize       */     0,
2357        /* offsetOfTypes */     offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2358        /* offsets       */     {(UINT16)(offsetof(PolicyLocality_In, locality))},
2359        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2360                                 TPMA_LOCALITY_P_UNMARSHAL,
2361                                 END_OF_LIST,
2362                                 END_OF_LIST}
2363    };
2364    #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2365    #else
2366    #define _PolicyLocalityDataAddress 0
2367    #endif // CC_PolicyLocality
2368    #if CC_PolicyNV
2369    #include "PolicyNV_fp.h"
2370    typedef TPM_RC  (PolicyNV_Entry)(
2371        PolicyNV_In                 *in
2372    );
2373    typedef const struct {
2374        PolicyNV_Entry          *entry;
2375        UINT16                  inSize;
2376        UINT16                  outSize;
2377        UINT16                  offsetOfTypes;
2378        UINT16                  paramOffsets[5];
2379        BYTE                    types[8];
2380    } PolicyNV_COMMAND_DESCRIPTOR_t;
2381    PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2382        /* entry         */     &TPM2_PolicyNV,
2383        /* inSize        */     (UINT16)(sizeof(PolicyNV_In)),
2384        /* outSize       */     0,
2385        /* offsetOfTypes */     offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2386        /* offsets       */     {(UINT16)(offsetof(PolicyNV_In, nvIndex)),
2387                                 (UINT16)(offsetof(PolicyNV_In, policySession)),
2388                                 (UINT16)(offsetof(PolicyNV_In, operandB)),
```

```
2389                                (UINT16)(offsetof(PolicyNV_In, offset)),
2390                                (UINT16)(offsetof(PolicyNV_In, operation))},
2391        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2392                                TPMI_RH_NV_INDEX_H_UNMARSHAL,
2393                                TPMI_SH_POLICY_H_UNMARSHAL,
2394                                TPM2B_OPERAND_P_UNMARSHAL,
2395                                UINT16_P_UNMARSHAL,
2396                                TPM_EO_P_UNMARSHAL,
2397                                END_OF_LIST,
2398                                END_OF_LIST}
2399    };
2400    #define _PolicyNVDataAddress (&_PolicyNVData)
2401    #else
2402    #define _PolicyNVDataAddress 0
2403    #endif // CC_PolicyNV
2404    #if CC_PolicyCounterTimer
2405    #include "PolicyCounterTimer_fp.h"
2406    typedef TPM_RC  (PolicyCounterTimer_Entry)(
2407        PolicyCounterTimer_In           *in
2408    );
2409    typedef const struct {
2410        PolicyCounterTimer_Entry    *entry;
2411        UINT16                      inSize;
2412        UINT16                      outSize;
2413        UINT16                      offsetOfTypes;
2414        UINT16                      paramOffsets[3];
2415        BYTE                        types[6];
2416    } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2417    PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2418        /* entry         */         &TPM2_PolicyCounterTimer,
2419        /* inSize        */         (UINT16)(sizeof(PolicyCounterTimer_In)),
2420        /* outSize       */         0,
2421        /* offsetOfTypes */         offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t,
    types),
2422        /* offsets       */         {(UINT16)(offsetof(PolicyCounterTimer_In, operandB)),
2423                                     (UINT16)(offsetof(PolicyCounterTimer_In, offset)),
2424                                     (UINT16)(offsetof(PolicyCounterTimer_In,
    operation))},
2425        /* types         */         {TPMI_SH_POLICY_H_UNMARSHAL,
2426                                     TPM2B_OPERAND_P_UNMARSHAL,
2427                                     UINT16_P_UNMARSHAL,
2428                                     TPM_EO_P_UNMARSHAL,
2429                                     END_OF_LIST,
2430                                     END_OF_LIST}
2431    };
2432    #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
2433    #else
2434    #define _PolicyCounterTimerDataAddress 0
2435    #endif // CC_PolicyCounterTimer
2436    #if CC_PolicyCommandCode
2437    #include "PolicyCommandCode_fp.h"
2438    typedef TPM_RC  (PolicyCommandCode_Entry)(
2439        PolicyCommandCode_In            *in
2440    );
2441    typedef const struct {
2442        PolicyCommandCode_Entry     *entry;
2443        UINT16                      inSize;
2444        UINT16                      outSize;
2445        UINT16                      offsetOfTypes;
2446        UINT16                      paramOffsets[1];
2447        BYTE                        types[4];
2448    } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
2449    PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
2450        /* entry         */         &TPM2_PolicyCommandCode,
2451        /* inSize        */         (UINT16)(sizeof(PolicyCommandCode_In)),
2452        /* outSize       */         0,
```

```
2453        /* offsetOfTypes */         offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t,
    types),
2454        /* offsets       */         {(UINT16)(offsetof(PolicyCommandCode_In, code))},
2455        /* types         */         {TPMI_SH_POLICY_H_UNMARSHAL,
2456                                     TPM_CC_P_UNMARSHAL,
2457                                     END_OF_LIST,
2458                                     END_OF_LIST}
2459    };
2460    #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
2461    #else
2462    #define _PolicyCommandCodeDataAddress 0
2463    #endif // CC_PolicyCommandCode
2464    #if CC_PolicyPhysicalPresence
2465    #include "PolicyPhysicalPresence_fp.h"
2466    typedef TPM_RC  (PolicyPhysicalPresence_Entry)(
2467        PolicyPhysicalPresence_In           *in
2468    );
2469    typedef const struct {
2470        PolicyPhysicalPresence_Entry    *entry;
2471        UINT16                          inSize;
2472        UINT16                          outSize;
2473        UINT16                          offsetOfTypes;
2474        BYTE                            types[3];
2475    } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
2476    PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
2477        /* entry         */             &TPM2_PolicyPhysicalPresence,
2478        /* inSize        */             (UINT16)(sizeof(PolicyPhysicalPresence_In)),
2479        /* outSize       */             0,
2480        /* offsetOfTypes */
    offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
2481        /* offsets       */             // No parameter offsets;
2482        /* types         */             {TPMI_SH_POLICY_H_UNMARSHAL,
2483                                         END_OF_LIST,
2484                                         END_OF_LIST}
2485    };
2486    #define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
2487    #else
2488    #define _PolicyPhysicalPresenceDataAddress 0
2489    #endif // CC_PolicyPhysicalPresence
2490    #if CC_PolicyCpHash
2491    #include "PolicyCpHash_fp.h"
2492    typedef TPM_RC  (PolicyCpHash_Entry)(
2493        PolicyCpHash_In             *in
2494    );
2495    typedef const struct {
2496        PolicyCpHash_Entry      *entry;
2497        UINT16                  inSize;
2498        UINT16                  outSize;
2499        UINT16                  offsetOfTypes;
2500        UINT16                  paramOffsets[1];
2501        BYTE                    types[4];
2502    } PolicyCpHash_COMMAND_DESCRIPTOR_t;
2503    PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
2504        /* entry         */     &TPM2_PolicyCpHash,
2505        /* inSize        */     (UINT16)(sizeof(PolicyCpHash_In)),
2506        /* outSize       */     0,
2507        /* offsetOfTypes */     offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
2508        /* offsets       */     {(UINT16)(offsetof(PolicyCpHash_In, cpHashA))},
2509        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2510                                 TPM2B_DIGEST_P_UNMARSHAL,
2511                                 END_OF_LIST,
2512                                 END_OF_LIST}
2513    };
2514    #define _PolicyCpHashDataAddress (&_PolicyCpHashData)
2515    #else
2516    #define _PolicyCpHashDataAddress 0
```

```
2517  #endif // CC_PolicyCpHash
2518  #if CC_PolicyNameHash
2519  #include "PolicyNameHash_fp.h"
2520  typedef TPM_RC  (PolicyNameHash_Entry)(
2521      PolicyNameHash_In           *in
2522  );
2523  typedef const struct {
2524      PolicyNameHash_Entry    *entry;
2525      UINT16                  inSize;
2526      UINT16                  outSize;
2527      UINT16                  offsetOfTypes;
2528      UINT16                  paramOffsets[1];
2529      BYTE                    types[4];
2530  } PolicyNameHash_COMMAND_DESCRIPTOR_t;
2531  PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
2532      /* entry         */     &TPM2_PolicyNameHash,
2533      /* inSize        */     (UINT16)(sizeof(PolicyNameHash_In)),
2534      /* outSize       */     0,
2535      /* offsetOfTypes */     offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
2536      /* offsets       */     {(UINT16)(offsetof(PolicyNameHash_In, nameHash))},
2537      /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2538                               TPM2B_DIGEST_P_UNMARSHAL,
2539                               END_OF_LIST,
2540                               END_OF_LIST}
2541  };
2542  #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
2543  #else
2544  #define _PolicyNameHashDataAddress 0
2545  #endif // CC_PolicyNameHash
2546  #if CC_PolicyDuplicationSelect
2547  #include "PolicyDuplicationSelect_fp.h"
2548  typedef TPM_RC  (PolicyDuplicationSelect_Entry)(
2549      PolicyDuplicationSelect_In          *in
2550  );
2551  typedef const struct {
2552      PolicyDuplicationSelect_Entry   *entry;
2553      UINT16                          inSize;
2554      UINT16                          outSize;
2555      UINT16                          offsetOfTypes;
2556      UINT16                          paramOffsets[3];
2557      BYTE                            types[6];
2558  } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
2559  PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
2560      /* entry         */             &TPM2_PolicyDuplicationSelect,
2561      /* inSize        */             (UINT16)(sizeof(PolicyDuplicationSelect_In)),
2562      /* outSize       */             0,
2563      /* offsetOfTypes */
      offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t, types),
2564      /* offsets       */             {(UINT16)(offsetof(PolicyDuplicationSelect_In,
      objectName)),
2565                                       (UINT16)(offsetof(PolicyDuplicationSelect_In,
      newParentName)),
2566                                       (UINT16)(offsetof(PolicyDuplicationSelect_In,
      includeObject))},
2567      /* types         */             {TPMI_SH_POLICY_H_UNMARSHAL,
2568                                       TPM2B_NAME_P_UNMARSHAL,
2569                                       TPM2B_NAME_P_UNMARSHAL,
2570                                       TPMI_YES_NO_P_UNMARSHAL,
2571                                       END_OF_LIST,
2572                                       END_OF_LIST}
2573  };
2574  #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
2575  #else
2576  #define _PolicyDuplicationSelectDataAddress 0
2577  #endif // CC_PolicyDuplicationSelect
2578  #if CC_PolicyAuthorize
```

```
2579    #include "PolicyAuthorize_fp.h"
2580    typedef TPM_RC  (PolicyAuthorize_Entry)(
2581        PolicyAuthorize_In          *in
2582    );
2583    typedef const struct {
2584        PolicyAuthorize_Entry   *entry;
2585        UINT16                  inSize;
2586        UINT16                  outSize;
2587        UINT16                  offsetOfTypes;
2588        UINT16                  paramOffsets[4];
2589        BYTE                    types[7];
2590    } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
2591    PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
2592        /* entry         */     &TPM2_PolicyAuthorize,
2593        /* inSize        */     (UINT16)(sizeof(PolicyAuthorize_In)),
2594        /* outSize       */     0,
2595        /* offsetOfTypes */     offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
2596        /* offsets       */     {(UINT16)(offsetof(PolicyAuthorize_In, approvedPolicy)),
2597                                 (UINT16)(offsetof(PolicyAuthorize_In, policyRef)),
2598                                 (UINT16)(offsetof(PolicyAuthorize_In, keySign)),
2599                                 (UINT16)(offsetof(PolicyAuthorize_In, checkTicket))},
2600        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2601                                 TPM2B_DIGEST_P_UNMARSHAL,
2602                                 TPM2B_NONCE_P_UNMARSHAL,
2603                                 TPM2B_NAME_P_UNMARSHAL,
2604                                 TPMT_TK_VERIFIED_P_UNMARSHAL,
2605                                 END_OF_LIST,
2606                                 END_OF_LIST}
2607    };
2608    #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
2609    #else
2610    #define _PolicyAuthorizeDataAddress 0
2611    #endif // CC_PolicyAuthorize
2612    #if CC_PolicyAuthValue
2613    #include "PolicyAuthValue_fp.h"
2614    typedef TPM_RC  (PolicyAuthValue_Entry)(
2615        PolicyAuthValue_In          *in
2616    );
2617    typedef const struct {
2618        PolicyAuthValue_Entry   *entry;
2619        UINT16                  inSize;
2620        UINT16                  outSize;
2621        UINT16                  offsetOfTypes;
2622        BYTE                    types[3];
2623    } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
2624    PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
2625        /* entry         */     &TPM2_PolicyAuthValue,
2626        /* inSize        */     (UINT16)(sizeof(PolicyAuthValue_In)),
2627        /* outSize       */     0,
2628        /* offsetOfTypes */     offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
2629        /* offsets       */     // No parameter offsets;
2630        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2631                                 END_OF_LIST,
2632                                 END_OF_LIST}
2633    };
2634    #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
2635    #else
2636    #define _PolicyAuthValueDataAddress 0
2637    #endif // CC_PolicyAuthValue
2638    #if CC_PolicyPassword
2639    #include "PolicyPassword_fp.h"
2640    typedef TPM_RC  (PolicyPassword_Entry)(
2641        PolicyPassword_In           *in
2642    );
2643    typedef const struct {
2644        PolicyPassword_Entry    *entry;
```

```
2645        UINT16                  inSize;
2646        UINT16                  outSize;
2647        UINT16                  offsetOfTypes;
2648        BYTE                    types[3];
2649    } PolicyPassword_COMMAND_DESCRIPTOR_t;
2650    PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
2651        /* entry          */    &TPM2_PolicyPassword,
2652        /* inSize         */    (UINT16)(sizeof(PolicyPassword_In)),
2653        /* outSize        */    0,
2654        /* offsetOfTypes */     offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
2655        /* offsets        */    // No parameter offsets;
2656        /* types          */    {TPMI_SH_POLICY_H_UNMARSHAL,
2657                                 END_OF_LIST,
2658                                 END_OF_LIST}
2659    };
2660    #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
2661    #else
2662    #define _PolicyPasswordDataAddress 0
2663    #endif // CC_PolicyPassword
2664    #if CC_PolicyGetDigest
2665    #include "PolicyGetDigest_fp.h"
2666    typedef TPM_RC  (PolicyGetDigest_Entry)(
2667        PolicyGetDigest_In          *in,
2668        PolicyGetDigest_Out         *out
2669    );
2670    typedef const struct {
2671        PolicyGetDigest_Entry   *entry;
2672        UINT16                  inSize;
2673        UINT16                  outSize;
2674        UINT16                  offsetOfTypes;
2675        BYTE                    types[4];
2676    } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
2677    PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
2678        /* entry          */    &TPM2_PolicyGetDigest,
2679        /* inSize         */    (UINT16)(sizeof(PolicyGetDigest_In)),
2680        /* outSize        */    (UINT16)(sizeof(PolicyGetDigest_Out)),
2681        /* offsetOfTypes */     offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
2682        /* offsets        */    // No parameter offsets;
2683        /* types          */    {TPMI_SH_POLICY_H_UNMARSHAL,
2684                                 END_OF_LIST,
2685                                 TPM2B_DIGEST_P_MARSHAL,
2686                                 END_OF_LIST}
2687    };
2688    #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
2689    #else
2690    #define _PolicyGetDigestDataAddress 0
2691    #endif // CC_PolicyGetDigest
2692    #if CC_PolicyNvWritten
2693    #include "PolicyNvWritten_fp.h"
2694    typedef TPM_RC  (PolicyNvWritten_Entry)(
2695        PolicyNvWritten_In          *in
2696    );
2697    typedef const struct {
2698        PolicyNvWritten_Entry   *entry;
2699        UINT16                  inSize;
2700        UINT16                  outSize;
2701        UINT16                  offsetOfTypes;
2702        UINT16                  paramOffsets[1];
2703        BYTE                    types[4];
2704    } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
2705    PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
2706        /* entry          */    &TPM2_PolicyNvWritten,
2707        /* inSize         */    (UINT16)(sizeof(PolicyNvWritten_In)),
2708        /* outSize        */    0,
2709        /* offsetOfTypes */     offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
2710        /* offsets        */    {(UINT16)(offsetof(PolicyNvWritten_In, writtenSet))},
```

```
2711        /* types          */     {TPMI_SH_POLICY_H_UNMARSHAL,
2712                                   TPMI_YES_NO_P_UNMARSHAL,
2713                                   END_OF_LIST,
2714                                   END_OF_LIST}
2715    };
2716    #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
2717    #else
2718    #define _PolicyNvWrittenDataAddress 0
2719    #endif // CC_PolicyNvWritten
2720    #if CC_PolicyTemplate
2721    #include "PolicyTemplate_fp.h"
2722    typedef TPM_RC  (PolicyTemplate_Entry)(
2723        PolicyTemplate_In           *in
2724    );
2725    typedef const struct {
2726        PolicyTemplate_Entry    *entry;
2727        UINT16                  inSize;
2728        UINT16                  outSize;
2729        UINT16                  offsetOfTypes;
2730        UINT16                  paramOffsets[1];
2731        BYTE                    types[4];
2732    } PolicyTemplate_COMMAND_DESCRIPTOR_t;
2733    PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
2734        /* entry          */     &TPM2_PolicyTemplate,
2735        /* inSize         */     (UINT16)(sizeof(PolicyTemplate_In)),
2736        /* outSize        */     0,
2737        /* offsetOfTypes */     offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
2738        /* offsets        */     {(UINT16)(offsetof(PolicyTemplate_In, templateHash))},
2739        /* types          */     {TPMI_SH_POLICY_H_UNMARSHAL,
2740                                   TPM2B_DIGEST_P_UNMARSHAL,
2741                                   END_OF_LIST,
2742                                   END_OF_LIST}
2743    };
2744    #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
2745    #else
2746    #define _PolicyTemplateDataAddress 0
2747    #endif // CC_PolicyTemplate
2748    #if CC_PolicyAuthorizeNV
2749    #include "PolicyAuthorizeNV_fp.h"
2750    typedef TPM_RC  (PolicyAuthorizeNV_Entry)(
2751        PolicyAuthorizeNV_In            *in
2752    );
2753    typedef const struct {
2754        PolicyAuthorizeNV_Entry     *entry;
2755        UINT16                      inSize;
2756        UINT16                      outSize;
2757        UINT16                      offsetOfTypes;
2758        UINT16                      paramOffsets[2];
2759        BYTE                        types[5];
2760    } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;
2761    PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t _PolicyAuthorizeNVData = {
2762        /* entry          */         &TPM2_PolicyAuthorizeNV,
2763        /* inSize         */         (UINT16)(sizeof(PolicyAuthorizeNV_In)),
2764        /* outSize        */         0,
2765        /* offsetOfTypes */         offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t,
    types),
2766        /* offsets        */         {(UINT16)(offsetof(PolicyAuthorizeNV_In, nvIndex)),
2767                                     (UINT16)(offsetof(PolicyAuthorizeNV_In,
    policySession))},
2768        /* types          */         {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2769                                     TPMI_RH_NV_INDEX_H_UNMARSHAL,
2770                                     TPMI_SH_POLICY_H_UNMARSHAL,
2771                                     END_OF_LIST,
2772                                     END_OF_LIST}
2773    };
2774    #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
```

```
#else
#define _PolicyAuthorizeNVDataAddress 0
#endif // CC_PolicyAuthorizeNV
#if CC_CreatePrimary
#include "CreatePrimary_fp.h"
typedef TPM_RC  (CreatePrimary_Entry)(
    CreatePrimary_In            *in,
    CreatePrimary_Out           *out
);
typedef const struct {
    CreatePrimary_Entry     *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[9];
    BYTE                    types[13];
} CreatePrimary_COMMAND_DESCRIPTOR_t;
CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
    /* entry         */     &TPM2_CreatePrimary,
    /* inSize        */     (UINT16)(sizeof(CreatePrimary_In)),
    /* outSize       */     (UINT16)(sizeof(CreatePrimary_Out)),
    /* offsetOfTypes */     offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(CreatePrimary_In, inSensitive)),
                             (UINT16)(offsetof(CreatePrimary_In, inPublic)),
                             (UINT16)(offsetof(CreatePrimary_In, outsideInfo)),
                             (UINT16)(offsetof(CreatePrimary_In, creationPCR)),
                             (UINT16)(offsetof(CreatePrimary_Out, outPublic)),
                             (UINT16)(offsetof(CreatePrimary_Out, creationData)),
                             (UINT16)(offsetof(CreatePrimary_Out, creationHash)),
                             (UINT16)(offsetof(CreatePrimary_Out, creationTicket)),
                             (UINT16)(offsetof(CreatePrimary_Out, name))},
    /* types         */     {TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
                             TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
                             TPM2B_PUBLIC_P_UNMARSHAL,
                             TPM2B_DATA_P_UNMARSHAL,
                             TPML_PCR_SELECTION_P_UNMARSHAL,
                             END_OF_LIST,
                             TPM_HANDLE_H_MARSHAL,
                             TPM2B_PUBLIC_P_MARSHAL,
                             TPM2B_CREATION_DATA_P_MARSHAL,
                             TPM2B_DIGEST_P_MARSHAL,
                             TPMT_TK_CREATION_P_MARSHAL,
                             TPM2B_NAME_P_MARSHAL,
                             END_OF_LIST}
};
#define _CreatePrimaryDataAddress (&_CreatePrimaryData)
#else
#define _CreatePrimaryDataAddress 0
#endif // CC_CreatePrimary
#if CC_HierarchyControl
#include "HierarchyControl_fp.h"
typedef TPM_RC  (HierarchyControl_Entry)(
    HierarchyControl_In         *in
);
typedef const struct {
    HierarchyControl_Entry  *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[2];
    BYTE                    types[5];
} HierarchyControl_COMMAND_DESCRIPTOR_t;
HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
    /* entry         */     &TPM2_HierarchyControl,
    /* inSize        */     (UINT16)(sizeof(HierarchyControl_In)),
    /* outSize       */     0,
```

```
2841        /* offsetOfTypes */     offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
2842        /* offsets       */     {(UINT16)(offsetof(HierarchyControl_In, enable)),
2843                                  (UINT16)(offsetof(HierarchyControl_In, state))},
2844        /* types         */     {TPMI_RH_HIERARCHY_H_UNMARSHAL,
2845                                  TPMI_RH_ENABLES_P_UNMARSHAL,
2846                                  TPMI_YES_NO_P_UNMARSHAL,
2847                                  END_OF_LIST,
2848                                  END_OF_LIST}
2849    };
2850    #define _HierarchyControlDataAddress (&_HierarchyControlData)
2851    #else
2852    #define _HierarchyControlDataAddress 0
2853    #endif // CC_HierarchyControl
2854    #if CC_SetPrimaryPolicy
2855    #include "SetPrimaryPolicy_fp.h"
2856    typedef TPM_RC  (SetPrimaryPolicy_Entry)(
2857        SetPrimaryPolicy_In         *in
2858    );
2859    typedef const struct {
2860        SetPrimaryPolicy_Entry  *entry;
2861        UINT16                  inSize;
2862        UINT16                  outSize;
2863        UINT16                  offsetOfTypes;
2864        UINT16                  paramOffsets[2];
2865        BYTE                    types[5];
2866    } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
2867    SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
2868        /* entry         */     &TPM2_SetPrimaryPolicy,
2869        /* inSize        */     (UINT16)(sizeof(SetPrimaryPolicy_In)),
2870        /* outSize       */     0,
2871        /* offsetOfTypes */     offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),
2872        /* offsets       */     {(UINT16)(offsetof(SetPrimaryPolicy_In, authPolicy)),
2873                                  (UINT16)(offsetof(SetPrimaryPolicy_In, hashAlg))},
2874        /* types         */     {TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL,
2875                                  TPM2B_DIGEST_P_UNMARSHAL,
2876                                  TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2877                                  END_OF_LIST,
2878                                  END_OF_LIST}
2879    };
2880    #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
2881    #else
2882    #define _SetPrimaryPolicyDataAddress 0
2883    #endif // CC_SetPrimaryPolicy
2884    #if CC_ChangePPS
2885    #include "ChangePPS_fp.h"
2886    typedef TPM_RC  (ChangePPS_Entry)(
2887        ChangePPS_In                *in
2888    );
2889    typedef const struct {
2890        ChangePPS_Entry         *entry;
2891        UINT16                  inSize;
2892        UINT16                  outSize;
2893        UINT16                  offsetOfTypes;
2894        BYTE                    types[3];
2895    } ChangePPS_COMMAND_DESCRIPTOR_t;
2896    ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
2897        /* entry         */     &TPM2_ChangePPS,
2898        /* inSize        */     (UINT16)(sizeof(ChangePPS_In)),
2899        /* outSize       */     0,
2900        /* offsetOfTypes */     offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
2901        /* offsets       */     // No parameter offsets;
2902        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
2903                                  END_OF_LIST,
2904                                  END_OF_LIST}
2905    };
2906    #define _ChangePPSDataAddress (&_ChangePPSData)
```

```
#else
#define _ChangePPSDataAddress 0
#endif // CC_ChangePPS
#if CC_ChangeEPS
#include "ChangeEPS_fp.h"
typedef TPM_RC  (ChangeEPS_Entry)(
    ChangeEPS_In                *in
);
typedef const struct {
    ChangeEPS_Entry         *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    BYTE                    types[3];
} ChangeEPS_COMMAND_DESCRIPTOR_t;
ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
    /* entry         */     &TPM2_ChangeEPS,
    /* inSize        */     (UINT16)(sizeof(ChangeEPS_In)),
    /* outSize       */     0,
    /* offsetOfTypes */     offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     // No parameter offsets;
    /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
                             END_OF_LIST,
                             END_OF_LIST}
};
#define _ChangeEPSDataAddress (&_ChangeEPSData)
#else
#define _ChangeEPSDataAddress 0
#endif // CC_ChangeEPS
#if CC_Clear
#include "Clear_fp.h"
typedef TPM_RC  (Clear_Entry)(
    Clear_In                    *in
);
typedef const struct {
    Clear_Entry             *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    BYTE                    types[3];
} Clear_COMMAND_DESCRIPTOR_t;
Clear_COMMAND_DESCRIPTOR_t _ClearData = {
    /* entry         */     &TPM2_Clear,
    /* inSize        */     (UINT16)(sizeof(Clear_In)),
    /* outSize       */     0,
    /* offsetOfTypes */     offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     // No parameter offsets;
    /* types         */     {TPMI_RH_CLEAR_H_UNMARSHAL,
                             END_OF_LIST,
                             END_OF_LIST}
};
#define _ClearDataAddress (&_ClearData)
#else
#define _ClearDataAddress 0
#endif // CC_Clear
#if CC_ClearControl
#include "ClearControl_fp.h"
typedef TPM_RC  (ClearControl_Entry)(
    ClearControl_In             *in
);
typedef const struct {
    ClearControl_Entry      *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[1];
```

```
2973        BYTE                    types[4];
2974    } ClearControl_COMMAND_DESCRIPTOR_t;
2975    ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
2976        /* entry         */     &TPM2_ClearControl,
2977        /* inSize        */     (UINT16)(sizeof(ClearControl_In)),
2978        /* outSize       */     0,
2979        /* offsetOfTypes */     offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),
2980        /* offsets       */     {(UINT16)(offsetof(ClearControl_In, disable))},
2981        /* types         */     {TPMI_RH_CLEAR_H_UNMARSHAL,
2982                                 TPMI_YES_NO_P_UNMARSHAL,
2983                                 END_OF_LIST,
2984                                 END_OF_LIST}
2985    };
2986    #define _ClearControlDataAddress (&_ClearControlData)
2987    #else
2988    #define _ClearControlDataAddress 0
2989    #endif // CC_ClearControl
2990    #if CC_HierarchyChangeAuth
2991    #include "HierarchyChangeAuth_fp.h"
2992    typedef TPM_RC  (HierarchyChangeAuth_Entry)(
2993        HierarchyChangeAuth_In          *in
2994    );
2995    typedef const struct {
2996        HierarchyChangeAuth_Entry   *entry;
2997        UINT16                      inSize;
2998        UINT16                      outSize;
2999        UINT16                      offsetOfTypes;
3000        UINT16                      paramOffsets[1];
3001        BYTE                        types[4];
3002    } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
3003    HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
3004        /* entry         */         &TPM2_HierarchyChangeAuth,
3005        /* inSize        */         (UINT16)(sizeof(HierarchyChangeAuth_In)),
3006        /* outSize       */         0,
3007        /* offsetOfTypes */         offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t,
    types),
3008        /* offsets       */         {(UINT16)(offsetof(HierarchyChangeAuth_In, newAuth))},
3009        /* types         */         {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
3010                                     TPM2B_AUTH_P_UNMARSHAL,
3011                                     END_OF_LIST,
3012                                     END_OF_LIST}
3013    };
3014    #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
3015    #else
3016    #define _HierarchyChangeAuthDataAddress 0
3017    #endif // CC_HierarchyChangeAuth
3018    #if CC_DictionaryAttackLockReset
3019    #include "DictionaryAttackLockReset_fp.h"
3020    typedef TPM_RC  (DictionaryAttackLockReset_Entry)(
3021        DictionaryAttackLockReset_In            *in
3022    );
3023    typedef const struct {
3024        DictionaryAttackLockReset_Entry     *entry;
3025        UINT16                              inSize;
3026        UINT16                              outSize;
3027        UINT16                              offsetOfTypes;
3028        BYTE                                types[3];
3029    } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
3030    DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
3031        /* entry         */                 &TPM2_DictionaryAttackLockReset,
3032        /* inSize        */
    (UINT16)(sizeof(DictionaryAttackLockReset_In)),
3033        /* outSize       */                 0,
3034        /* offsetOfTypes */
    offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
3035        /* offsets       */                 // No parameter offsets;
```

```
3036        /* types         */                 {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3037                                              END_OF_LIST,
3038                                              END_OF_LIST}
3039    };
3040    #define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
3041    #else
3042    #define _DictionaryAttackLockResetDataAddress 0
3043    #endif // CC_DictionaryAttackLockReset
3044    #if CC_DictionaryAttackParameters
3045    #include "DictionaryAttackParameters_fp.h"
3046    typedef TPM_RC  (DictionaryAttackParameters_Entry)(
3047        DictionaryAttackParameters_In           *in
3048    );
3049    typedef const struct {
3050        DictionaryAttackParameters_Entry    *entry;
3051        UINT16                              inSize;
3052        UINT16                              outSize;
3053        UINT16                              offsetOfTypes;
3054        UINT16                              paramOffsets[3];
3055        BYTE                                types[6];
3056    } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
3057    DictionaryAttackParameters_COMMAND_DESCRIPTOR_t _DictionaryAttackParametersData =
        {
3058        /* entry         */                 &TPM2_DictionaryAttackParameters,
3059        /* inSize        */
        (UINT16)(sizeof(DictionaryAttackParameters_In)),
3060        /* outSize       */                 0,
3061        /* offsetOfTypes */
        offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
3062        /* offsets       */
        {(UINT16)(offsetof(DictionaryAttackParameters_In, newMaxTries)),
3063
        (UINT16)(offsetof(DictionaryAttackParameters_In, newRecoveryTime)),
3064
        (UINT16)(offsetof(DictionaryAttackParameters_In, lockoutRecovery))},
3065        /* types         */                 {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3066                                              UINT32_P_UNMARSHAL,
3067                                              UINT32_P_UNMARSHAL,
3068                                              UINT32_P_UNMARSHAL,
3069                                              END_OF_LIST,
3070                                              END_OF_LIST}
3071    };
3072    #define _DictionaryAttackParametersDataAddress (&_DictionaryAttackParametersData)
3073    #else
3074    #define _DictionaryAttackParametersDataAddress 0
3075    #endif // CC_DictionaryAttackParameters
3076    #if CC_PP_Commands
3077    #include "PP_Commands_fp.h"
3078    typedef TPM_RC  (PP_Commands_Entry)(
3079        PP_Commands_In              *in
3080    );
3081    typedef const struct {
3082        PP_Commands_Entry       *entry;
3083        UINT16                  inSize;
3084        UINT16                  outSize;
3085        UINT16                  offsetOfTypes;
3086        UINT16                  paramOffsets[2];
3087        BYTE                    types[5];
3088    } PP_Commands_COMMAND_DESCRIPTOR_t;
3089    PP_Commands_COMMAND_DESCRIPTOR_t _PP_CommandsData = {
3090        /* entry         */     &TPM2_PP_Commands,
3091        /* inSize        */     (UINT16)(sizeof(PP_Commands_In)),
3092        /* outSize       */     0,
3093        /* offsetOfTypes */     offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
3094        /* offsets       */     {(UINT16)(offsetof(PP_Commands_In, setList)),
3095                                 (UINT16)(offsetof(PP_Commands_In, clearList))},
```

```
3096        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
3097                                  TPML_CC_P_UNMARSHAL,
3098                                  TPML_CC_P_UNMARSHAL,
3099                                  END_OF_LIST,
3100                                  END_OF_LIST}
3101    };
3102    #define _PP_CommandsDataAddress (&_PP_CommandsData)
3103    #else
3104    #define _PP_CommandsDataAddress 0
3105    #endif // CC_PP_Commands
3106    #if CC_SetAlgorithmSet
3107    #include "SetAlgorithmSet_fp.h"
3108    typedef TPM_RC  (SetAlgorithmSet_Entry)(
3109        SetAlgorithmSet_In          *in
3110    );
3111    typedef const struct {
3112        SetAlgorithmSet_Entry   *entry;
3113        UINT16                  inSize;
3114        UINT16                  outSize;
3115        UINT16                  offsetOfTypes;
3116        UINT16                  paramOffsets[1];
3117        BYTE                    types[4];
3118    } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
3119    SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
3120        /* entry         */     &TPM2_SetAlgorithmSet,
3121        /* inSize        */     (UINT16)(sizeof(SetAlgorithmSet_In)),
3122        /* outSize       */     0,
3123        /* offsetOfTypes */     offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
3124        /* offsets       */     {(UINT16)(offsetof(SetAlgorithmSet_In, algorithmSet))},
3125        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
3126                                  UINT32_P_UNMARSHAL,
3127                                  END_OF_LIST,
3128                                  END_OF_LIST}
3129    };
3130    #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
3131    #else
3132    #define _SetAlgorithmSetDataAddress 0
3133    #endif // CC_SetAlgorithmSet
3134    #if CC_FieldUpgradeStart
3135    #include "FieldUpgradeStart_fp.h"
3136    typedef TPM_RC  (FieldUpgradeStart_Entry)(
3137        FieldUpgradeStart_In            *in
3138    );
3139    typedef const struct {
3140        FieldUpgradeStart_Entry     *entry;
3141        UINT16                      inSize;
3142        UINT16                      outSize;
3143        UINT16                      offsetOfTypes;
3144        UINT16                      paramOffsets[3];
3145        BYTE                        types[6];
3146    } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
3147    FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
3148        /* entry         */         &TPM2_FieldUpgradeStart,
3149        /* inSize        */         (UINT16)(sizeof(FieldUpgradeStart_In)),
3150        /* outSize       */         0,
3151        /* offsetOfTypes */         offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t,
    types),
3152        /* offsets       */         {(UINT16)(offsetof(FieldUpgradeStart_In, keyHandle)),
3153                                     (UINT16)(offsetof(FieldUpgradeStart_In, fuDigest)),
3154                                     (UINT16)(offsetof(FieldUpgradeStart_In,
    manifestSignature))},
3155        /* types         */         {TPMI_RH_PLATFORM_H_UNMARSHAL,
3156                                     TPMI_DH_OBJECT_H_UNMARSHAL,
3157                                     TPM2B_DIGEST_P_UNMARSHAL,
3158                                     TPMT_SIGNATURE_P_UNMARSHAL,
3159                                     END_OF_LIST,
```

```
                                    END_OF_LIST}
};
#define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
#else
#define _FieldUpgradeStartDataAddress 0
#endif // CC_FieldUpgradeStart
#if CC_FieldUpgradeData
#include "FieldUpgradeData_fp.h"
typedef TPM_RC  (FieldUpgradeData_Entry)(
    FieldUpgradeData_In         *in,
    FieldUpgradeData_Out        *out
);
typedef const struct {
    FieldUpgradeData_Entry  *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[1];
    BYTE                    types[5];
} FieldUpgradeData_COMMAND_DESCRIPTOR_t;
FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
    /* entry         */     &TPM2_FieldUpgradeData,
    /* inSize        */     (UINT16)(sizeof(FieldUpgradeData_In)),
    /* outSize       */     (UINT16)(sizeof(FieldUpgradeData_Out)),
    /* offsetOfTypes */     offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(FieldUpgradeData_Out, firstDigest))},
    /* types         */     {TPM2B_MAX_BUFFER_P_UNMARSHAL,
                             END_OF_LIST,
                             TPMT_HA_P_MARSHAL,
                             TPMT_HA_P_MARSHAL,
                             END_OF_LIST}
};
#define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
#else
#define _FieldUpgradeDataDataAddress 0
#endif // CC_FieldUpgradeData
#if CC_FirmwareRead
#include "FirmwareRead_fp.h"
typedef TPM_RC  (FirmwareRead_Entry)(
    FirmwareRead_In             *in,
    FirmwareRead_Out            *out
);
typedef const struct {
    FirmwareRead_Entry      *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    BYTE                    types[4];
} FirmwareRead_COMMAND_DESCRIPTOR_t;
FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
    /* entry         */     &TPM2_FirmwareRead,
    /* inSize        */     (UINT16)(sizeof(FirmwareRead_In)),
    /* outSize       */     (UINT16)(sizeof(FirmwareRead_Out)),
    /* offsetOfTypes */     offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     // No parameter offsets;
    /* types         */     {UINT32_P_UNMARSHAL,
                             END_OF_LIST,
                             TPM2B_MAX_BUFFER_P_MARSHAL,
                             END_OF_LIST}
};
#define _FirmwareReadDataAddress (&_FirmwareReadData)
#else
#define _FirmwareReadDataAddress 0
#endif // CC_FirmwareRead
#if CC_ContextSave
#include "ContextSave_fp.h"
```

```
3226    typedef TPM_RC  (ContextSave_Entry)(
3227        ContextSave_In              *in,
3228        ContextSave_Out             *out
3229    );
3230    typedef const struct {
3231        ContextSave_Entry       *entry;
3232        UINT16                  inSize;
3233        UINT16                  outSize;
3234        UINT16                  offsetOfTypes;
3235        BYTE                    types[4];
3236    } ContextSave_COMMAND_DESCRIPTOR_t;
3237    ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
3238        /* entry         */     &TPM2_ContextSave,
3239        /* inSize        */     (UINT16)(sizeof(ContextSave_In)),
3240        /* outSize       */     (UINT16)(sizeof(ContextSave_Out)),
3241        /* offsetOfTypes */     offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
3242        /* offsets       */     // No parameter offsets;
3243        /* types         */     {TPMI_DH_CONTEXT_H_UNMARSHAL,
3244                                 END_OF_LIST,
3245                                 TPMS_CONTEXT_P_MARSHAL,
3246                                 END_OF_LIST}
3247    };
3248    #define _ContextSaveDataAddress (&_ContextSaveData)
3249    #else
3250    #define _ContextSaveDataAddress 0
3251    #endif // CC_ContextSave
3252    #if CC_ContextLoad
3253    #include "ContextLoad_fp.h"
3254    typedef TPM_RC  (ContextLoad_Entry)(
3255        ContextLoad_In              *in,
3256        ContextLoad_Out             *out
3257    );
3258    typedef const struct {
3259        ContextLoad_Entry       *entry;
3260        UINT16                  inSize;
3261        UINT16                  outSize;
3262        UINT16                  offsetOfTypes;
3263        BYTE                    types[4];
3264    } ContextLoad_COMMAND_DESCRIPTOR_t;
3265    ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
3266        /* entry         */     &TPM2_ContextLoad,
3267        /* inSize        */     (UINT16)(sizeof(ContextLoad_In)),
3268        /* outSize       */     (UINT16)(sizeof(ContextLoad_Out)),
3269        /* offsetOfTypes */     offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
3270        /* offsets       */     // No parameter offsets;
3271        /* types         */     {TPMS_CONTEXT_P_UNMARSHAL,
3272                                 END_OF_LIST,
3273                                 TPMI_DH_CONTEXT_H_MARSHAL,
3274                                 END_OF_LIST}
3275    };
3276    #define _ContextLoadDataAddress (&_ContextLoadData)
3277    #else
3278    #define _ContextLoadDataAddress 0
3279    #endif // CC_ContextLoad
3280    #if CC_FlushContext
3281    #include "FlushContext_fp.h"
3282    typedef TPM_RC  (FlushContext_Entry)(
3283        FlushContext_In             *in
3284    );
3285    typedef const struct {
3286        FlushContext_Entry      *entry;
3287        UINT16                  inSize;
3288        UINT16                  outSize;
3289        UINT16                  offsetOfTypes;
3290        BYTE                    types[3];
3291    } FlushContext_COMMAND_DESCRIPTOR_t;
```

```
3292  FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
3293      /* entry         */     &TPM2_FlushContext,
3294      /* inSize        */     (UINT16)(sizeof(FlushContext_In)),
3295      /* outSize       */     0,
3296      /* offsetOfTypes */     offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
3297      /* offsets       */     // No parameter offsets;
3298      /* types         */     {TPMI_DH_CONTEXT_P_UNMARSHAL,
3299                               END_OF_LIST,
3300                               END_OF_LIST}
3301  };
3302  #define _FlushContextDataAddress (&_FlushContextData)
3303  #else
3304  #define _FlushContextDataAddress 0
3305  #endif // CC_FlushContext
3306  #if CC_EvictControl
3307  #include "EvictControl_fp.h"
3308  typedef TPM_RC  (EvictControl_Entry)(
3309      EvictControl_In             *in
3310  );
3311  typedef const struct {
3312      EvictControl_Entry      *entry;
3313      UINT16                  inSize;
3314      UINT16                  outSize;
3315      UINT16                  offsetOfTypes;
3316      UINT16                  paramOffsets[2];
3317      BYTE                    types[5];
3318  } EvictControl_COMMAND_DESCRIPTOR_t;
3319  EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
3320      /* entry         */     &TPM2_EvictControl,
3321      /* inSize        */     (UINT16)(sizeof(EvictControl_In)),
3322      /* outSize       */     0,
3323      /* offsetOfTypes */     offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
3324      /* offsets       */     {(UINT16)(offsetof(EvictControl_In, objectHandle)),
3325                               (UINT16)(offsetof(EvictControl_In, persistentHandle))},
3326      /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3327                               TPMI_DH_OBJECT_H_UNMARSHAL,
3328                               TPMI_DH_PERSISTENT_P_UNMARSHAL,
3329                               END_OF_LIST,
3330                               END_OF_LIST}
3331  };
3332  #define _EvictControlDataAddress (&_EvictControlData)
3333  #else
3334  #define _EvictControlDataAddress 0
3335  #endif // CC_EvictControl
3336  #if CC_ReadClock
3337  #include "ReadClock_fp.h"
3338  typedef TPM_RC  (ReadClock_Entry)(
3339      ReadClock_Out               *out
3340  );
3341  typedef const struct {
3342      ReadClock_Entry         *entry;
3343      UINT16                  inSize;
3344      UINT16                  outSize;
3345      UINT16                  offsetOfTypes;
3346      BYTE                    types[3];
3347  } ReadClock_COMMAND_DESCRIPTOR_t;
3348  ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
3349      /* entry         */     &TPM2_ReadClock,
3350      /* inSize        */     0,
3351      /* outSize       */     (UINT16)(sizeof(ReadClock_Out)),
3352      /* offsetOfTypes */     offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),
3353      /* offsets       */     // No parameter offsets;
3354      /* types         */     {END_OF_LIST,
3355                               TPMS_TIME_INFO_P_MARSHAL,
3356                               END_OF_LIST}
3357  };
```

```
3358    #define _ReadClockDataAddress (&_ReadClockData)
3359    #else
3360    #define _ReadClockDataAddress 0
3361    #endif // CC_ReadClock
3362    #if CC_ClockSet
3363    #include "ClockSet_fp.h"
3364    typedef TPM_RC  (ClockSet_Entry)(
3365        ClockSet_In                 *in
3366    );
3367    typedef const struct {
3368        ClockSet_Entry          *entry;
3369        UINT16                  inSize;
3370        UINT16                  outSize;
3371        UINT16                  offsetOfTypes;
3372        UINT16                  paramOffsets[1];
3373        BYTE                    types[4];
3374    } ClockSet_COMMAND_DESCRIPTOR_t;
3375    ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
3376        /* entry         */     &TPM2_ClockSet,
3377        /* inSize        */     (UINT16)(sizeof(ClockSet_In)),
3378        /* outSize       */     0,
3379        /* offsetOfTypes */     offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
3380        /* offsets       */     {(UINT16)(offsetof(ClockSet_In, newTime))},
3381        /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3382                                 UINT64_P_UNMARSHAL,
3383                                 END_OF_LIST,
3384                                 END_OF_LIST}
3385    };
3386    #define _ClockSetDataAddress (&_ClockSetData)
3387    #else
3388    #define _ClockSetDataAddress 0
3389    #endif // CC_ClockSet
3390    #if CC_ClockRateAdjust
3391    #include "ClockRateAdjust_fp.h"
3392    typedef TPM_RC  (ClockRateAdjust_Entry)(
3393        ClockRateAdjust_In          *in
3394    );
3395    typedef const struct {
3396        ClockRateAdjust_Entry   *entry;
3397        UINT16                  inSize;
3398        UINT16                  outSize;
3399        UINT16                  offsetOfTypes;
3400        UINT16                  paramOffsets[1];
3401        BYTE                    types[4];
3402    } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
3403    ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
3404        /* entry         */     &TPM2_ClockRateAdjust,
3405        /* inSize        */     (UINT16)(sizeof(ClockRateAdjust_In)),
3406        /* outSize       */     0,
3407        /* offsetOfTypes */     offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
3408        /* offsets       */     {(UINT16)(offsetof(ClockRateAdjust_In, rateAdjust))},
3409        /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3410                                 TPM_CLOCK_ADJUST_P_UNMARSHAL,
3411                                 END_OF_LIST,
3412                                 END_OF_LIST}
3413    };
3414    #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
3415    #else
3416    #define _ClockRateAdjustDataAddress 0
3417    #endif // CC_ClockRateAdjust
3418    #if CC_GetCapability
3419    #include "GetCapability_fp.h"
3420    typedef TPM_RC  (GetCapability_Entry)(
3421        GetCapability_In            *in,
3422        GetCapability_Out           *out
3423    );
```

```
3424    typedef const struct {
3425        GetCapability_Entry     *entry;
3426        UINT16                  inSize;
3427        UINT16                  outSize;
3428        UINT16                  offsetOfTypes;
3429        UINT16                  paramOffsets[3];
3430        BYTE                    types[7];
3431    } GetCapability_COMMAND_DESCRIPTOR_t;
3432    GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
3433        /* entry         */     &TPM2_GetCapability,
3434        /* inSize        */     (UINT16)(sizeof(GetCapability_In)),
3435        /* outSize       */     (UINT16)(sizeof(GetCapability_Out)),
3436        /* offsetOfTypes */     offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
3437        /* offsets       */     {(UINT16)(offsetof(GetCapability_In, property)),
3438                                 (UINT16)(offsetof(GetCapability_In, propertyCount)),
3439                                 (UINT16)(offsetof(GetCapability_Out, capabilityData))},
3440        /* types         */     {TPM_CAP_P_UNMARSHAL,
3441                                 UINT32_P_UNMARSHAL,
3442                                 UINT32_P_UNMARSHAL,
3443                                 END_OF_LIST,
3444                                 TPMI_YES_NO_P_MARSHAL,
3445                                 TPMS_CAPABILITY_DATA_P_MARSHAL,
3446                                 END_OF_LIST}
3447    };
3448    #define _GetCapabilityDataAddress (&_GetCapabilityData)
3449    #else
3450    #define _GetCapabilityDataAddress 0
3451    #endif // CC_GetCapability
3452    #if CC_TestParms
3453    #include "TestParms_fp.h"
3454    typedef TPM_RC  (TestParms_Entry)(
3455        TestParms_In                *in
3456    );
3457    typedef const struct {
3458        TestParms_Entry         *entry;
3459        UINT16                  inSize;
3460        UINT16                  outSize;
3461        UINT16                  offsetOfTypes;
3462        BYTE                    types[3];
3463    } TestParms_COMMAND_DESCRIPTOR_t;
3464    TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
3465        /* entry         */     &TPM2_TestParms,
3466        /* inSize        */     (UINT16)(sizeof(TestParms_In)),
3467        /* outSize       */     0,
3468        /* offsetOfTypes */     offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
3469        /* offsets       */     // No parameter offsets;
3470        /* types         */     {TPMT_PUBLIC_PARMS_P_UNMARSHAL,
3471                                 END_OF_LIST,
3472                                 END_OF_LIST}
3473    };
3474    #define _TestParmsDataAddress (&_TestParmsData)
3475    #else
3476    #define _TestParmsDataAddress 0
3477    #endif // CC_TestParms
3478    #if CC_NV_DefineSpace
3479    #include "NV_DefineSpace_fp.h"
3480    typedef TPM_RC  (NV_DefineSpace_Entry)(
3481        NV_DefineSpace_In           *in
3482    );
3483    typedef const struct {
3484        NV_DefineSpace_Entry    *entry;
3485        UINT16                  inSize;
3486        UINT16                  outSize;
3487        UINT16                  offsetOfTypes;
3488        UINT16                  paramOffsets[2];
3489        BYTE                    types[5];
```

```
3490  } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
3491  NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
3492      /* entry          */     &TPM2_NV_DefineSpace,
3493      /* inSize         */     (UINT16)(sizeof(NV_DefineSpace_In)),
3494      /* outSize        */     0,
3495      /* offsetOfTypes */     offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
3496      /* offsets        */     {(UINT16)(offsetof(NV_DefineSpace_In, auth)),
3497                                (UINT16)(offsetof(NV_DefineSpace_In, publicInfo))},
3498      /* types          */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3499                                TPM2B_AUTH_P_UNMARSHAL,
3500                                TPM2B_NV_PUBLIC_P_UNMARSHAL,
3501                                END_OF_LIST,
3502                                END_OF_LIST}
3503  };
3504  #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
3505  #else
3506  #define _NV_DefineSpaceDataAddress 0
3507  #endif // CC_NV_DefineSpace
3508  #if CC_NV_UndefineSpace
3509  #include "NV_UndefineSpace_fp.h"
3510  typedef TPM_RC  (NV_UndefineSpace_Entry)(
3511      NV_UndefineSpace_In         *in
3512  );
3513  typedef const struct {
3514      NV_UndefineSpace_Entry  *entry;
3515      UINT16                  inSize;
3516      UINT16                  outSize;
3517      UINT16                  offsetOfTypes;
3518      UINT16                  paramOffsets[1];
3519      BYTE                    types[4];
3520  } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
3521  NV_UndefineSpace_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceData = {
3522      /* entry          */     &TPM2_NV_UndefineSpace,
3523      /* inSize         */     (UINT16)(sizeof(NV_UndefineSpace_In)),
3524      /* outSize        */     0,
3525      /* offsetOfTypes */     offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t, types),
3526      /* offsets        */     {(UINT16)(offsetof(NV_UndefineSpace_In, nvIndex))},
3527      /* types          */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3528                                TPMI_RH_NV_INDEX_H_UNMARSHAL,
3529                                END_OF_LIST,
3530                                END_OF_LIST}
3531  };
3532  #define _NV_UndefineSpaceDataAddress (&_NV_UndefineSpaceData)
3533  #else
3534  #define _NV_UndefineSpaceDataAddress 0
3535  #endif // CC_NV_UndefineSpace
3536  #if CC_NV_UndefineSpaceSpecial
3537  #include "NV_UndefineSpaceSpecial_fp.h"
3538  typedef TPM_RC  (NV_UndefineSpaceSpecial_Entry)(
3539      NV_UndefineSpaceSpecial_In          *in
3540  );
3541  typedef const struct {
3542      NV_UndefineSpaceSpecial_Entry   *entry;
3543      UINT16                          inSize;
3544      UINT16                          outSize;
3545      UINT16                          offsetOfTypes;
3546      UINT16                          paramOffsets[1];
3547      BYTE                            types[4];
3548  } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
3549  NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceSpecialData = {
3550      /* entry          */             &TPM2_NV_UndefineSpaceSpecial,
3551      /* inSize         */             (UINT16)(sizeof(NV_UndefineSpaceSpecial_In)),
3552      /* outSize        */             0,
3553      /* offsetOfTypes */
      offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
```



```
3554        /* offsets       */             {(UINT16)(offsetof(NV_UndefineSpaceSpecial_In,
    platform))},
3555        /* types         */             {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3556                                         TPMI_RH_PLATFORM_H_UNMARSHAL,
3557                                         END_OF_LIST,
3558                                         END_OF_LIST}
3559    };
3560    #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
3561    #else
3562    #define _NV_UndefineSpaceSpecialDataAddress 0
3563    #endif // CC_NV_UndefineSpaceSpecial
3564    #if CC_NV_ReadPublic
3565    #include "NV_ReadPublic_fp.h"
3566    typedef TPM_RC  (NV_ReadPublic_Entry)(
3567        NV_ReadPublic_In            *in,
3568        NV_ReadPublic_Out           *out
3569    );
3570    typedef const struct {
3571        NV_ReadPublic_Entry     *entry;
3572        UINT16                  inSize;
3573        UINT16                  outSize;
3574        UINT16                  offsetOfTypes;
3575        UINT16                  paramOffsets[1];
3576        BYTE                    types[5];
3577    } NV_ReadPublic_COMMAND_DESCRIPTOR_t;
3578    NV_ReadPublic_COMMAND_DESCRIPTOR_t _NV_ReadPublicData = {
3579        /* entry         */     &TPM2_NV_ReadPublic,
3580        /* inSize        */     (UINT16)(sizeof(NV_ReadPublic_In)),
3581        /* outSize       */     (UINT16)(sizeof(NV_ReadPublic_Out)),
3582        /* offsetOfTypes */     offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
3583        /* offsets       */     {(UINT16)(offsetof(NV_ReadPublic_Out, nvName))},
3584        /* types         */     {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3585                                 END_OF_LIST,
3586                                 TPM2B_NV_PUBLIC_P_MARSHAL,
3587                                 TPM2B_NAME_P_MARSHAL,
3588                                 END_OF_LIST}
3589    };
3590    #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
3591    #else
3592    #define _NV_ReadPublicDataAddress 0
3593    #endif // CC_NV_ReadPublic
3594    #if CC_NV_Write
3595    #include "NV_Write_fp.h"
3596    typedef TPM_RC  (NV_Write_Entry)(
3597        NV_Write_In                 *in
3598    );
3599    typedef const struct {
3600        NV_Write_Entry          *entry;
3601        UINT16                  inSize;
3602        UINT16                  outSize;
3603        UINT16                  offsetOfTypes;
3604        UINT16                  paramOffsets[3];
3605        BYTE                    types[6];
3606    } NV_Write_COMMAND_DESCRIPTOR_t;
3607    NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
3608        /* entry         */     &TPM2_NV_Write,
3609        /* inSize        */     (UINT16)(sizeof(NV_Write_In)),
3610        /* outSize       */     0,
3611        /* offsetOfTypes */     offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
3612        /* offsets       */     {(UINT16)(offsetof(NV_Write_In, nvIndex)),
3613                                 (UINT16)(offsetof(NV_Write_In, data)),
3614                                 (UINT16)(offsetof(NV_Write_In, offset))},
3615        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3616                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3617                                 TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3618                                 UINT16_P_UNMARSHAL,
```

```
3619                                END_OF_LIST,
3620                                END_OF_LIST}
3621    };
3622    #define _NV_WriteDataAddress (&_NV_WriteData)
3623    #else
3624    #define _NV_WriteDataAddress 0
3625    #endif // CC_NV_Write
3626    #if CC_NV_Increment
3627    #include "NV_Increment_fp.h"
3628    typedef TPM_RC  (NV_Increment_Entry)(
3629        NV_Increment_In             *in
3630    );
3631    typedef const struct {
3632        NV_Increment_Entry      *entry;
3633        UINT16                  inSize;
3634        UINT16                  outSize;
3635        UINT16                  offsetOfTypes;
3636        UINT16                  paramOffsets[1];
3637        BYTE                    types[4];
3638    } NV_Increment_COMMAND_DESCRIPTOR_t;
3639    NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
3640        /* entry         */     &TPM2_NV_Increment,
3641        /* inSize        */     (UINT16)(sizeof(NV_Increment_In)),
3642        /* outSize       */     0,
3643        /* offsetOfTypes */     offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),
3644        /* offsets       */     {(UINT16)(offsetof(NV_Increment_In, nvIndex))},
3645        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3646                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3647                                 END_OF_LIST,
3648                                 END_OF_LIST}
3649    };
3650    #define _NV_IncrementDataAddress (&_NV_IncrementData)
3651    #else
3652    #define _NV_IncrementDataAddress 0
3653    #endif // CC_NV_Increment
3654    #if CC_NV_Extend
3655    #include "NV_Extend_fp.h"
3656    typedef TPM_RC  (NV_Extend_Entry)(
3657        NV_Extend_In                *in
3658    );
3659    typedef const struct {
3660        NV_Extend_Entry         *entry;
3661        UINT16                  inSize;
3662        UINT16                  outSize;
3663        UINT16                  offsetOfTypes;
3664        UINT16                  paramOffsets[2];
3665        BYTE                    types[5];
3666    } NV_Extend_COMMAND_DESCRIPTOR_t;
3667    NV_Extend_COMMAND_DESCRIPTOR_t _NV_ExtendData = {
3668        /* entry         */     &TPM2_NV_Extend,
3669        /* inSize        */     (UINT16)(sizeof(NV_Extend_In)),
3670        /* outSize       */     0,
3671        /* offsetOfTypes */     offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
3672        /* offsets       */     {(UINT16)(offsetof(NV_Extend_In, nvIndex)),
3673                                 (UINT16)(offsetof(NV_Extend_In, data))},
3674        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3675                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3676                                 TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3677                                 END_OF_LIST,
3678                                 END_OF_LIST}
3679    };
3680    #define _NV_ExtendDataAddress (&_NV_ExtendData)
3681    #else
3682    #define _NV_ExtendDataAddress 0
3683    #endif // CC_NV_Extend
3684    #if CC_NV_SetBits
```

```
3685  #include "NV_SetBits_fp.h"
3686  typedef TPM_RC  (NV_SetBits_Entry)(
3687      NV_SetBits_In               *in
3688  );
3689  typedef const struct {
3690      NV_SetBits_Entry        *entry;
3691      UINT16                  inSize;
3692      UINT16                  outSize;
3693      UINT16                  offsetOfTypes;
3694      UINT16                  paramOffsets[2];
3695      BYTE                    types[5];
3696  } NV_SetBits_COMMAND_DESCRIPTOR_t;
3697  NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
3698      /* entry         */     &TPM2_NV_SetBits,
3699      /* inSize        */     (UINT16)(sizeof(NV_SetBits_In)),
3700      /* outSize       */     0,
3701      /* offsetOfTypes */     offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
3702      /* offsets       */     {(UINT16)(offsetof(NV_SetBits_In, nvIndex)),
3703                               (UINT16)(offsetof(NV_SetBits_In, bits))},
3704      /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3705                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
3706                               UINT64_P_UNMARSHAL,
3707                               END_OF_LIST,
3708                               END_OF_LIST}
3709  };
3710  #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
3711  #else
3712  #define _NV_SetBitsDataAddress 0
3713  #endif // CC_NV_SetBits
3714  #if CC_NV_WriteLock
3715  #include "NV_WriteLock_fp.h"
3716  typedef TPM_RC  (NV_WriteLock_Entry)(
3717      NV_WriteLock_In             *in
3718  );
3719  typedef const struct {
3720      NV_WriteLock_Entry      *entry;
3721      UINT16                  inSize;
3722      UINT16                  outSize;
3723      UINT16                  offsetOfTypes;
3724      UINT16                  paramOffsets[1];
3725      BYTE                    types[4];
3726  } NV_WriteLock_COMMAND_DESCRIPTOR_t;
3727  NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
3728      /* entry         */     &TPM2_NV_WriteLock,
3729      /* inSize        */     (UINT16)(sizeof(NV_WriteLock_In)),
3730      /* outSize       */     0,
3731      /* offsetOfTypes */     offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
3732      /* offsets       */     {(UINT16)(offsetof(NV_WriteLock_In, nvIndex))},
3733      /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3734                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
3735                               END_OF_LIST,
3736                               END_OF_LIST}
3737  };
3738  #define _NV_WriteLockDataAddress (&_NV_WriteLockData)
3739  #else
3740  #define _NV_WriteLockDataAddress 0
3741  #endif // CC_NV_WriteLock
3742  #if CC_NV_GlobalWriteLock
3743  #include "NV_GlobalWriteLock_fp.h"
3744  typedef TPM_RC  (NV_GlobalWriteLock_Entry)(
3745      NV_GlobalWriteLock_In           *in
3746  );
3747  typedef const struct {
3748      NV_GlobalWriteLock_Entry    *entry;
3749      UINT16                      inSize;
3750      UINT16                      outSize;
```

```
3751        UINT16                      offsetOfTypes;
3752        BYTE                        types[3];
3753    } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
3754    NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
3755        /* entry         */         &TPM2_NV_GlobalWriteLock,
3756        /* inSize        */         (UINT16)(sizeof(NV_GlobalWriteLock_In)),
3757        /* outSize       */         0,
3758        /* offsetOfTypes */         offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t,
    types),
3759        /* offsets       */         // No parameter offsets;
3760        /* types         */         {TPMI_RH_PROVISION_H_UNMARSHAL,
3761                                     END_OF_LIST,
3762                                     END_OF_LIST}
3763    };
3764    #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
3765    #else
3766    #define _NV_GlobalWriteLockDataAddress 0
3767    #endif // CC_NV_GlobalWriteLock
3768    #if CC_NV_Read
3769    #include "NV_Read_fp.h"
3770    typedef TPM_RC  (NV_Read_Entry)(
3771        NV_Read_In                  *in,
3772        NV_Read_Out                 *out
3773    );
3774    typedef const struct {
3775        NV_Read_Entry           *entry;
3776        UINT16                  inSize;
3777        UINT16                  outSize;
3778        UINT16                  offsetOfTypes;
3779        UINT16                  paramOffsets[3];
3780        BYTE                    types[7];
3781    } NV_Read_COMMAND_DESCRIPTOR_t;
3782    NV_Read_COMMAND_DESCRIPTOR_t _NV_ReadData = {
3783        /* entry         */     &TPM2_NV_Read,
3784        /* inSize        */     (UINT16)(sizeof(NV_Read_In)),
3785        /* outSize       */     (UINT16)(sizeof(NV_Read_Out)),
3786        /* offsetOfTypes */     offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
3787        /* offsets       */     {(UINT16)(offsetof(NV_Read_In, nvIndex)),
3788                                 (UINT16)(offsetof(NV_Read_In, size)),
3789                                 (UINT16)(offsetof(NV_Read_In, offset))},
3790        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3791                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3792                                 UINT16_P_UNMARSHAL,
3793                                 UINT16_P_UNMARSHAL,
3794                                 END_OF_LIST,
3795                                 TPM2B_MAX_NV_BUFFER_P_MARSHAL,
3796                                 END_OF_LIST}
3797    };
3798    #define _NV_ReadDataAddress (&_NV_ReadData)
3799    #else
3800    #define _NV_ReadDataAddress 0
3801    #endif // CC_NV_Read
3802    #if CC_NV_ReadLock
3803    #include "NV_ReadLock_fp.h"
3804    typedef TPM_RC  (NV_ReadLock_Entry)(
3805        NV_ReadLock_In              *in
3806    );
3807    typedef const struct {
3808        NV_ReadLock_Entry       *entry;
3809        UINT16                  inSize;
3810        UINT16                  outSize;
3811        UINT16                  offsetOfTypes;
3812        UINT16                  paramOffsets[1];
3813        BYTE                    types[4];
3814    } NV_ReadLock_COMMAND_DESCRIPTOR_t;
3815    NV_ReadLock_COMMAND_DESCRIPTOR_t _NV_ReadLockData = {
```

```
3816        /* entry          */     &TPM2_NV_ReadLock,
3817        /* inSize         */     (UINT16)(sizeof(NV_ReadLock_In)),
3818        /* outSize        */     0,
3819        /* offsetOfTypes  */     offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
3820        /* offsets        */     {(UINT16)(offsetof(NV_ReadLock_In, nvIndex))},
3821        /* types          */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3822                                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
3823                                  END_OF_LIST,
3824                                  END_OF_LIST}
3825    };
3826    #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
3827    #else
3828    #define _NV_ReadLockDataAddress 0
3829    #endif // CC_NV_ReadLock
3830    #if CC_NV_ChangeAuth
3831    #include "NV_ChangeAuth_fp.h"
3832    typedef TPM_RC  (NV_ChangeAuth_Entry)(
3833        NV_ChangeAuth_In            *in
3834    );
3835    typedef const struct {
3836        NV_ChangeAuth_Entry     *entry;
3837        UINT16                  inSize;
3838        UINT16                  outSize;
3839        UINT16                  offsetOfTypes;
3840        UINT16                  paramOffsets[1];
3841        BYTE                    types[4];
3842    } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
3843    NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
3844        /* entry          */     &TPM2_NV_ChangeAuth,
3845        /* inSize         */     (UINT16)(sizeof(NV_ChangeAuth_In)),
3846        /* outSize        */     0,
3847        /* offsetOfTypes  */     offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
3848        /* offsets        */     {(UINT16)(offsetof(NV_ChangeAuth_In, newAuth))},
3849        /* types          */     {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3850                                  TPM2B_AUTH_P_UNMARSHAL,
3851                                  END_OF_LIST,
3852                                  END_OF_LIST}
3853    };
3854    #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
3855    #else
3856    #define _NV_ChangeAuthDataAddress 0
3857    #endif // CC_NV_ChangeAuth
3858    #if CC_NV_Certify
3859    #include "NV_Certify_fp.h"
3860    typedef TPM_RC  (NV_Certify_Entry)(
3861        NV_Certify_In               *in,
3862        NV_Certify_Out              *out
3863    );
3864    typedef const struct {
3865        NV_Certify_Entry        *entry;
3866        UINT16                  inSize;
3867        UINT16                  outSize;
3868        UINT16                  offsetOfTypes;
3869        UINT16                  paramOffsets[7];
3870        BYTE                    types[11];
3871    } NV_Certify_COMMAND_DESCRIPTOR_t;
3872    NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
3873        /* entry          */     &TPM2_NV_Certify,
3874        /* inSize         */     (UINT16)(sizeof(NV_Certify_In)),
3875        /* outSize        */     (UINT16)(sizeof(NV_Certify_Out)),
3876        /* offsetOfTypes  */     offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
3877        /* offsets        */     {(UINT16)(offsetof(NV_Certify_In, authHandle)),
3878                                  (UINT16)(offsetof(NV_Certify_In, nvIndex)),
3879                                  (UINT16)(offsetof(NV_Certify_In, qualifyingData)),
3880                                  (UINT16)(offsetof(NV_Certify_In, inScheme)),
3881                                  (UINT16)(offsetof(NV_Certify_In, size)),
```

```
3882                                (UINT16)(offsetof(NV_Certify_In, offset)),
3883                                (UINT16)(offsetof(NV_Certify_Out, signature))},
3884        /* types          */    {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
3885                                TPMI_RH_NV_AUTH_H_UNMARSHAL,
3886                                TPMI_RH_NV_INDEX_H_UNMARSHAL,
3887                                TPM2B_DATA_P_UNMARSHAL,
3888                                TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
3889                                UINT16_P_UNMARSHAL,
3890                                UINT16_P_UNMARSHAL,
3891                                END_OF_LIST,
3892                                TPM2B_ATTEST_P_MARSHAL,
3893                                TPMT_SIGNATURE_P_MARSHAL,
3894                                END_OF_LIST}
3895    };
3896    #define _NV_CertifyDataAddress (&_NV_CertifyData)
3897    #else
3898    #define _NV_CertifyDataAddress 0
3899    #endif // CC_NV_Certify
3900    #if CC_AC_GetCapability
3901    #include "AC_GetCapability_fp.h"
3902    typedef TPM_RC  (AC_GetCapability_Entry)(
3903        AC_GetCapability_In         *in,
3904        AC_GetCapability_Out        *out
3905    );
3906    typedef const struct {
3907        AC_GetCapability_Entry  *entry;
3908        UINT16                  inSize;
3909        UINT16                  outSize;
3910        UINT16                  offsetOfTypes;
3911        UINT16                  paramOffsets[3];
3912        BYTE                    types[7];
3913    } AC_GetCapability_COMMAND_DESCRIPTOR_t;
3914    AC_GetCapability_COMMAND_DESCRIPTOR_t _AC_GetCapabilityData = {
3915        /* entry          */    &TPM2_AC_GetCapability,
3916        /* inSize         */    (UINT16)(sizeof(AC_GetCapability_In)),
3917        /* outSize        */    (UINT16)(sizeof(AC_GetCapability_Out)),
3918        /* offsetOfTypes */    offsetof(AC_GetCapability_COMMAND_DESCRIPTOR_t, types),
3919        /* offsets        */    {(UINT16)(offsetof(AC_GetCapability_In, capability)),
3920                                (UINT16)(offsetof(AC_GetCapability_In, count)),
3921                                (UINT16)(offsetof(AC_GetCapability_Out,
    capabilitiesData))},
3922        /* types          */    {TPMI_RH_AC_H_UNMARSHAL,
3923                                TPM_AT_P_UNMARSHAL,
3924                                UINT32_P_UNMARSHAL,
3925                                END_OF_LIST,
3926                                TPMI_YES_NO_P_MARSHAL,
3927                                TPML_AC_CAPABILITIES_P_MARSHAL,
3928                                END_OF_LIST}
3929    };
3930    #define _AC_GetCapabilityDataAddress (&_AC_GetCapabilityData)
3931    #else
3932    #define _AC_GetCapabilityDataAddress 0
3933    #endif // CC_AC_GetCapability
3934    #if CC_AC_Send
3935    #include "AC_Send_fp.h"
3936    typedef TPM_RC  (AC_Send_Entry)(
3937        AC_Send_In                  *in,
3938        AC_Send_Out                 *out
3939    );
3940    typedef const struct {
3941        AC_Send_Entry           *entry;
3942        UINT16                  inSize;
3943        UINT16                  outSize;
3944        UINT16                  offsetOfTypes;
3945        UINT16                  paramOffsets[3];
3946        BYTE                    types[7];
```

```
3947  } AC_Send_COMMAND_DESCRIPTOR_t;
3948  AC_Send_COMMAND_DESCRIPTOR_t _AC_SendData = {
3949      /* entry          */     &TPM2_AC_Send,
3950      /* inSize         */     (UINT16)(sizeof(AC_Send_In)),
3951      /* outSize        */     (UINT16)(sizeof(AC_Send_Out)),
3952      /* offsetOfTypes */     offsetof(AC_Send_COMMAND_DESCRIPTOR_t, types),
3953      /* offsets        */     {(UINT16)(offsetof(AC_Send_In, authHandle)),
3954                                (UINT16)(offsetof(AC_Send_In, ac)),
3955                                (UINT16)(offsetof(AC_Send_In, acDataIn))},
3956      /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
3957                                TPMI_RH_NV_AUTH_H_UNMARSHAL,
3958                                TPMI_RH_AC_H_UNMARSHAL,
3959                                TPM2B_MAX_BUFFER_P_UNMARSHAL,
3960                                END_OF_LIST,
3961                                TPMS_AC_OUTPUT_P_MARSHAL,
3962                                END_OF_LIST}
3963  };
3964  #define _AC_SendDataAddress (&_AC_SendData)
3965  #else
3966  #define _AC_SendDataAddress 0
3967  #endif // CC_AC_Send
3968  #if CC_Policy_AC_SendSelect
3969  #include "Policy_AC_SendSelect_fp.h"
3970  typedef TPM_RC  (Policy_AC_SendSelect_Entry)(
3971      Policy_AC_SendSelect_In         *in
3972  );
3973  typedef const struct {
3974      Policy_AC_SendSelect_Entry  *entry;
3975      UINT16                      inSize;
3976      UINT16                      outSize;
3977      UINT16                      offsetOfTypes;
3978      UINT16                      paramOffsets[4];
3979      BYTE                        types[7];
3980  } Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;
3981  Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
3982      /* entry          */         &TPM2_Policy_AC_SendSelect,
3983      /* inSize         */         (UINT16)(sizeof(Policy_AC_SendSelect_In)),
3984      /* outSize        */         0,
3985      /* offsetOfTypes */         offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t,
      types),
3986      /* offsets        */         {(UINT16)(offsetof(Policy_AC_SendSelect_In,
      objectName)),
3987                                   (UINT16)(offsetof(Policy_AC_SendSelect_In,
      authHandleName)),
3988                                   (UINT16)(offsetof(Policy_AC_SendSelect_In, acName)),
3989                                   (UINT16)(offsetof(Policy_AC_SendSelect_In,
      includeObject))},
3990      /* types          */         {TPMI_SH_POLICY_H_UNMARSHAL,
3991                                   TPM2B_NAME_P_UNMARSHAL,
3992                                   TPM2B_NAME_P_UNMARSHAL,
3993                                   TPM2B_NAME_P_UNMARSHAL,
3994                                   TPMI_YES_NO_P_UNMARSHAL,
3995                                   END_OF_LIST,
3996                                   END_OF_LIST}
3997  };
3998  #define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
3999  #else
4000  #define _Policy_AC_SendSelectDataAddress 0
4001  #endif // CC_Policy_AC_SendSelect
4002  #if CC_ACT_SetTimeout
4003  #include "ACT_SetTimeout_fp.h"
4004  typedef TPM_RC  (ACT_SetTimeout_Entry)(
4005      ACT_SetTimeout_In           *in
4006  );
4007  typedef const struct {
4008      ACT_SetTimeout_Entry    *entry;
```

```
4009        UINT16                  inSize;
4010        UINT16                  outSize;
4011        UINT16                  offsetOfTypes;
4012        UINT16                  paramOffsets[1];
4013        BYTE                    types[4];
4014    } ACT_SetTimeout_COMMAND_DESCRIPTOR_t;
4015    ACT_SetTimeout_COMMAND_DESCRIPTOR_t _ACT_SetTimeoutData = {
4016        /* entry         */     &TPM2_ACT_SetTimeout,
4017        /* inSize        */     (UINT16)(sizeof(ACT_SetTimeout_In)),
4018        /* outSize       */     0,
4019        /* offsetOfTypes */     offsetof(ACT_SetTimeout_COMMAND_DESCRIPTOR_t, types),
4020        /* offsets       */     {(UINT16)(offsetof(ACT_SetTimeout_In, startTimeout))},
4021        /* types         */     {TPMI_RH_ACT_H_UNMARSHAL,
4022                                 UINT32_P_UNMARSHAL,
4023                                 END_OF_LIST,
4024                                 END_OF_LIST}
4025    };
4026    #define _ACT_SetTimeoutDataAddress (&_ACT_SetTimeoutData)
4027    #else
4028    #define _ACT_SetTimeoutDataAddress 0
4029    #endif // CC_ACT_SetTimeout
4030    #if CC_Vendor_TCG_Test
4031    #include "Vendor_TCG_Test_fp.h"
4032    typedef TPM_RC  (Vendor_TCG_Test_Entry)(
4033        Vendor_TCG_Test_In          *in,
4034        Vendor_TCG_Test_Out         *out
4035    );
4036    typedef const struct {
4037        Vendor_TCG_Test_Entry   *entry;
4038        UINT16                  inSize;
4039        UINT16                  outSize;
4040        UINT16                  offsetOfTypes;
4041        BYTE                    types[4];
4042    } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
4043    Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
4044        /* entry         */     &TPM2_Vendor_TCG_Test,
4045        /* inSize        */     (UINT16)(sizeof(Vendor_TCG_Test_In)),
4046        /* outSize       */     (UINT16)(sizeof(Vendor_TCG_Test_Out)),
4047        /* offsetOfTypes */     offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
4048        /* offsets       */     // No parameter offsets;
4049        /* types         */     {TPM2B_DATA_P_UNMARSHAL,
4050                                 END_OF_LIST,
4051                                 TPM2B_DATA_P_MARSHAL,
4052                                 END_OF_LIST}
4053    };
4054    #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
4055    #else
4056    #define _Vendor_TCG_TestDataAddress 0
4057    #endif // CC_Vendor_TCG_Test
4058    COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
4059    #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
4060            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceSpecialDataAddress,
4061    #endif // CC_NV_UndefineSpaceSpecial
4062    #if (PAD_LIST || CC_EvictControl)
4063            (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
4064    #endif // CC_EvictControl
4065    #if (PAD_LIST || CC_HierarchyControl)
4066            (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
4067    #endif // CC_HierarchyControl
4068    #if (PAD_LIST || CC_NV_UndefineSpace)
4069            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceDataAddress,
4070    #endif // CC_NV_UndefineSpace
4071    #if (PAD_LIST)
4072            (COMMAND_DESCRIPTOR_t *)0,
4073    #endif //
4074    #if (PAD_LIST || CC_ChangeEPS)
```

```
4075            (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
4076    #endif // CC_ChangeEPS
4077    #if (PAD_LIST || CC_ChangePPS)
4078            (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
4079    #endif // CC_ChangePPS
4080    #if (PAD_LIST || CC_Clear)
4081            (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
4082    #endif // CC_Clear
4083    #if (PAD_LIST || CC_ClearControl)
4084            (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
4085    #endif // CC_ClearControl
4086    #if (PAD_LIST || CC_ClockSet)
4087            (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
4088    #endif // CC_ClockSet
4089    #if (PAD_LIST || CC_HierarchyChangeAuth)
4090            (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
4091    #endif // CC_HierarchyChangeAuth
4092    #if (PAD_LIST || CC_NV_DefineSpace)
4093            (COMMAND_DESCRIPTOR_t *)_NV_DefineSpaceDataAddress,
4094    #endif // CC_NV_DefineSpace
4095    #if (PAD_LIST || CC_PCR_Allocate)
4096            (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
4097    #endif // CC_PCR_Allocate
4098    #if (PAD_LIST || CC_PCR_SetAuthPolicy)
4099            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
4100    #endif // CC_PCR_SetAuthPolicy
4101    #if (PAD_LIST || CC_PP_Commands)
4102            (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
4103    #endif // CC_PP_Commands
4104    #if (PAD_LIST || CC_SetPrimaryPolicy)
4105            (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
4106    #endif // CC_SetPrimaryPolicy
4107    #if (PAD_LIST || CC_FieldUpgradeStart)
4108            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
4109    #endif // CC_FieldUpgradeStart
4110    #if (PAD_LIST || CC_ClockRateAdjust)
4111            (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
4112    #endif // CC_ClockRateAdjust
4113    #if (PAD_LIST || CC_CreatePrimary)
4114            (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
4115    #endif // CC_CreatePrimary
4116    #if (PAD_LIST || CC_NV_GlobalWriteLock)
4117            (COMMAND_DESCRIPTOR_t *)_NV_GlobalWriteLockDataAddress,
4118    #endif // CC_NV_GlobalWriteLock
4119    #if (PAD_LIST || CC_GetCommandAuditDigest)
4120            (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
4121    #endif // CC_GetCommandAuditDigest
4122    #if (PAD_LIST || CC_NV_Increment)
4123            (COMMAND_DESCRIPTOR_t *)_NV_IncrementDataAddress,
4124    #endif // CC_NV_Increment
4125    #if (PAD_LIST || CC_NV_SetBits)
4126            (COMMAND_DESCRIPTOR_t *)_NV_SetBitsDataAddress,
4127    #endif // CC_NV_SetBits
4128    #if (PAD_LIST || CC_NV_Extend)
4129            (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
4130    #endif // CC_NV_Extend
4131    #if (PAD_LIST || CC_NV_Write)
4132            (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
4133    #endif // CC_NV_Write
4134    #if (PAD_LIST || CC_NV_WriteLock)
4135            (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
4136    #endif // CC_NV_WriteLock
4137    #if (PAD_LIST || CC_DictionaryAttackLockReset)
4138            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
4139    #endif // CC_DictionaryAttackLockReset
4140    #if (PAD_LIST || CC_DictionaryAttackParameters)
```

```
4141            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
4142    #endif // CC_DictionaryAttackParameters
4143    #if (PAD_LIST || CC_NV_ChangeAuth)
4144            (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
4145    #endif // CC_NV_ChangeAuth
4146    #if (PAD_LIST || CC_PCR_Event)
4147            (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
4148    #endif // CC_PCR_Event
4149    #if (PAD_LIST || CC_PCR_Reset)
4150            (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
4151    #endif // CC_PCR_Reset
4152    #if (PAD_LIST || CC_SequenceComplete)
4153            (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
4154    #endif // CC_SequenceComplete
4155    #if (PAD_LIST || CC_SetAlgorithmSet)
4156            (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
4157    #endif // CC_SetAlgorithmSet
4158    #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
4159            (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
4160    #endif // CC_SetCommandCodeAuditStatus
4161    #if (PAD_LIST || CC_FieldUpgradeData)
4162            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
4163    #endif // CC_FieldUpgradeData
4164    #if (PAD_LIST || CC_IncrementalSelfTest)
4165            (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
4166    #endif // CC_IncrementalSelfTest
4167    #if (PAD_LIST || CC_SelfTest)
4168            (COMMAND_DESCRIPTOR_t *)_SelfTestDataAddress,
4169    #endif // CC_SelfTest
4170    #if (PAD_LIST || CC_Startup)
4171            (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
4172    #endif // CC_Startup
4173    #if (PAD_LIST || CC_Shutdown)
4174            (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
4175    #endif // CC_Shutdown
4176    #if (PAD_LIST || CC_StirRandom)
4177            (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
4178    #endif // CC_StirRandom
4179    #if (PAD_LIST || CC_ActivateCredential)
4180            (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
4181    #endif // CC_ActivateCredential
4182    #if (PAD_LIST || CC_Certify)
4183            (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
4184    #endif // CC_Certify
4185    #if (PAD_LIST || CC_PolicyNV)
4186            (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
4187    #endif // CC_PolicyNV
4188    #if (PAD_LIST || CC_CertifyCreation)
4189            (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
4190    #endif // CC_CertifyCreation
4191    #if (PAD_LIST || CC_Duplicate)
4192            (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
4193    #endif // CC_Duplicate
4194    #if (PAD_LIST || CC_GetTime)
4195            (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
4196    #endif // CC_GetTime
4197    #if (PAD_LIST || CC_GetSessionAuditDigest)
4198            (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
4199    #endif // CC_GetSessionAuditDigest
4200    #if (PAD_LIST || CC_NV_Read)
4201            (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
4202    #endif // CC_NV_Read
4203    #if (PAD_LIST || CC_NV_ReadLock)
4204            (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
4205    #endif // CC_NV_ReadLock
4206    #if (PAD_LIST || CC_ObjectChangeAuth)
```

```
4207            (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
4208    #endif // CC_ObjectChangeAuth
4209    #if (PAD_LIST || CC_PolicySecret)
4210            (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
4211    #endif // CC_PolicySecret
4212    #if (PAD_LIST || CC_Rewrap)
4213            (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
4214    #endif // CC_Rewrap
4215    #if (PAD_LIST || CC_Create)
4216            (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
4217    #endif // CC_Create
4218    #if (PAD_LIST || CC_ECDH_ZGen)
4219            (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
4220    #endif // CC_ECDH_ZGen
4221    #if (PAD_LIST || (CC_HMAC || CC_MAC))
4222    #    if CC_HMAC
4223            (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
4224    #    endif
4225    #    if CC_MAC
4226            (COMMAND_DESCRIPTOR_t *)_MACDataAddress,
4227    #    endif
4228    #    if (CC_HMAC || CC_MAC) > 1
4229    #        error "More than one aliased command defined"
4230    #    endif
4231    #endif // CC_HMAC CC_MAC
4232    #if (PAD_LIST || CC_Import)
4233            (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
4234    #endif // CC_Import
4235    #if (PAD_LIST || CC_Load)
4236            (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
4237    #endif // CC_Load
4238    #if (PAD_LIST || CC_Quote)
4239            (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
4240    #endif // CC_Quote
4241    #if (PAD_LIST || CC_RSA_Decrypt)
4242            (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
4243    #endif // CC_RSA_Decrypt
4244    #if (PAD_LIST)
4245            (COMMAND_DESCRIPTOR_t *)0,
4246    #endif //
4247    #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
4248    #    if CC_HMAC_Start
4249            (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
4250    #    endif
4251    #    if CC_MAC_Start
4252            (COMMAND_DESCRIPTOR_t *)_MAC_StartDataAddress,
4253    #    endif
4254    #    if (CC_HMAC_Start || CC_MAC_Start) > 1
4255    #        error "More than one aliased command defined"
4256    #    endif
4257    #endif // CC_HMAC_Start CC_MAC_Start
4258    #if (PAD_LIST || CC_SequenceUpdate)
4259            (COMMAND_DESCRIPTOR_t *)_SequenceUpdateDataAddress,
4260    #endif // CC_SequenceUpdate
4261    #if (PAD_LIST || CC_Sign)
4262            (COMMAND_DESCRIPTOR_t *)_SignDataAddress,
4263    #endif // CC_Sign
4264    #if (PAD_LIST || CC_Unseal)
4265            (COMMAND_DESCRIPTOR_t *)_UnsealDataAddress,
4266    #endif // CC_Unseal
4267    #if (PAD_LIST)
4268            (COMMAND_DESCRIPTOR_t *)0,
4269    #endif //
4270    #if (PAD_LIST || CC_PolicySigned)
4271            (COMMAND_DESCRIPTOR_t *)_PolicySignedDataAddress,
4272    #endif // CC_PolicySigned
```

```
4273    #if (PAD_LIST || CC_ContextLoad)
4274            (COMMAND_DESCRIPTOR_t *)_ContextLoadDataAddress,
4275    #endif // CC_ContextLoad
4276    #if (PAD_LIST || CC_ContextSave)
4277            (COMMAND_DESCRIPTOR_t *)_ContextSaveDataAddress,
4278    #endif // CC_ContextSave
4279    #if (PAD_LIST || CC_ECDH_KeyGen)
4280            (COMMAND_DESCRIPTOR_t *)_ECDH_KeyGenDataAddress,
4281    #endif // CC_ECDH_KeyGen
4282    #if (PAD_LIST || CC_EncryptDecrypt)
4283            (COMMAND_DESCRIPTOR_t *)_EncryptDecryptDataAddress,
4284    #endif // CC_EncryptDecrypt
4285    #if (PAD_LIST || CC_FlushContext)
4286            (COMMAND_DESCRIPTOR_t *)_FlushContextDataAddress,
4287    #endif // CC_FlushContext
4288    #if (PAD_LIST)
4289            (COMMAND_DESCRIPTOR_t *)0,
4290    #endif //
4291    #if (PAD_LIST || CC_LoadExternal)
4292            (COMMAND_DESCRIPTOR_t *)_LoadExternalDataAddress,
4293    #endif // CC_LoadExternal
4294    #if (PAD_LIST || CC_MakeCredential)
4295            (COMMAND_DESCRIPTOR_t *)_MakeCredentialDataAddress,
4296    #endif // CC_MakeCredential
4297    #if (PAD_LIST || CC_NV_ReadPublic)
4298            (COMMAND_DESCRIPTOR_t *)_NV_ReadPublicDataAddress,
4299    #endif // CC_NV_ReadPublic
4300    #if (PAD_LIST || CC_PolicyAuthorize)
4301            (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeDataAddress,
4302    #endif // CC_PolicyAuthorize
4303    #if (PAD_LIST || CC_PolicyAuthValue)
4304            (COMMAND_DESCRIPTOR_t *)_PolicyAuthValueDataAddress,
4305    #endif // CC_PolicyAuthValue
4306    #if (PAD_LIST || CC_PolicyCommandCode)
4307            (COMMAND_DESCRIPTOR_t *)_PolicyCommandCodeDataAddress,
4308    #endif // CC_PolicyCommandCode
4309    #if (PAD_LIST || CC_PolicyCounterTimer)
4310            (COMMAND_DESCRIPTOR_t *)_PolicyCounterTimerDataAddress,
4311    #endif // CC_PolicyCounterTimer
4312    #if (PAD_LIST || CC_PolicyCpHash)
4313            (COMMAND_DESCRIPTOR_t *)_PolicyCpHashDataAddress,
4314    #endif // CC_PolicyCpHash
4315    #if (PAD_LIST || CC_PolicyLocality)
4316            (COMMAND_DESCRIPTOR_t *)_PolicyLocalityDataAddress,
4317    #endif // CC_PolicyLocality
4318    #if (PAD_LIST || CC_PolicyNameHash)
4319            (COMMAND_DESCRIPTOR_t *)_PolicyNameHashDataAddress,
4320    #endif // CC_PolicyNameHash
4321    #if (PAD_LIST || CC_PolicyOR)
4322            (COMMAND_DESCRIPTOR_t *)_PolicyORDataAddress,
4323    #endif // CC_PolicyOR
4324    #if (PAD_LIST || CC_PolicyTicket)
4325            (COMMAND_DESCRIPTOR_t *)_PolicyTicketDataAddress,
4326    #endif // CC_PolicyTicket
4327    #if (PAD_LIST || CC_ReadPublic)
4328            (COMMAND_DESCRIPTOR_t *)_ReadPublicDataAddress,
4329    #endif // CC_ReadPublic
4330    #if (PAD_LIST || CC_RSA_Encrypt)
4331            (COMMAND_DESCRIPTOR_t *)_RSA_EncryptDataAddress,
4332    #endif // CC_RSA_Encrypt
4333    #if (PAD_LIST)
4334            (COMMAND_DESCRIPTOR_t *)0,
4335    #endif //
4336    #if (PAD_LIST || CC_StartAuthSession)
4337            (COMMAND_DESCRIPTOR_t *)_StartAuthSessionDataAddress,
4338    #endif // CC_StartAuthSession
```

```
4339    #if (PAD_LIST || CC_VerifySignature)
4340            (COMMAND_DESCRIPTOR_t *)_VerifySignatureDataAddress,
4341    #endif // CC_VerifySignature
4342    #if (PAD_LIST || CC_ECC_Parameters)
4343            (COMMAND_DESCRIPTOR_t *)_ECC_ParametersDataAddress,
4344    #endif // CC_ECC_Parameters
4345    #if (PAD_LIST || CC_FirmwareRead)
4346            (COMMAND_DESCRIPTOR_t *)_FirmwareReadDataAddress,
4347    #endif // CC_FirmwareRead
4348    #if (PAD_LIST || CC_GetCapability)
4349            (COMMAND_DESCRIPTOR_t *)_GetCapabilityDataAddress,
4350    #endif // CC_GetCapability
4351    #if (PAD_LIST || CC_GetRandom)
4352            (COMMAND_DESCRIPTOR_t *)_GetRandomDataAddress,
4353    #endif // CC_GetRandom
4354    #if (PAD_LIST || CC_GetTestResult)
4355            (COMMAND_DESCRIPTOR_t *)_GetTestResultDataAddress,
4356    #endif // CC_GetTestResult
4357    #if (PAD_LIST || CC_Hash)
4358            (COMMAND_DESCRIPTOR_t *)_HashDataAddress,
4359    #endif // CC_Hash
4360    #if (PAD_LIST || CC_PCR_Read)
4361            (COMMAND_DESCRIPTOR_t *)_PCR_ReadDataAddress,
4362    #endif // CC_PCR_Read
4363    #if (PAD_LIST || CC_PolicyPCR)
4364            (COMMAND_DESCRIPTOR_t *)_PolicyPCRDataAddress,
4365    #endif // CC_PolicyPCR
4366    #if (PAD_LIST || CC_PolicyRestart)
4367            (COMMAND_DESCRIPTOR_t *)_PolicyRestartDataAddress,
4368    #endif // CC_PolicyRestart
4369    #if (PAD_LIST || CC_ReadClock)
4370            (COMMAND_DESCRIPTOR_t *)_ReadClockDataAddress,
4371    #endif // CC_ReadClock
4372    #if (PAD_LIST || CC_PCR_Extend)
4373            (COMMAND_DESCRIPTOR_t *)_PCR_ExtendDataAddress,
4374    #endif // CC_PCR_Extend
4375    #if (PAD_LIST || CC_PCR_SetAuthValue)
4376            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthValueDataAddress,
4377    #endif // CC_PCR_SetAuthValue
4378    #if (PAD_LIST || CC_NV_Certify)
4379            (COMMAND_DESCRIPTOR_t *)_NV_CertifyDataAddress,
4380    #endif // CC_NV_Certify
4381    #if (PAD_LIST || CC_EventSequenceComplete)
4382            (COMMAND_DESCRIPTOR_t *)_EventSequenceCompleteDataAddress,
4383    #endif // CC_EventSequenceComplete
4384    #if (PAD_LIST || CC_HashSequenceStart)
4385            (COMMAND_DESCRIPTOR_t *)_HashSequenceStartDataAddress,
4386    #endif // CC_HashSequenceStart
4387    #if (PAD_LIST || CC_PolicyPhysicalPresence)
4388            (COMMAND_DESCRIPTOR_t *)_PolicyPhysicalPresenceDataAddress,
4389    #endif // CC_PolicyPhysicalPresence
4390    #if (PAD_LIST || CC_PolicyDuplicationSelect)
4391            (COMMAND_DESCRIPTOR_t *)_PolicyDuplicationSelectDataAddress,
4392    #endif // CC_PolicyDuplicationSelect
4393    #if (PAD_LIST || CC_PolicyGetDigest)
4394            (COMMAND_DESCRIPTOR_t *)_PolicyGetDigestDataAddress,
4395    #endif // CC_PolicyGetDigest
4396    #if (PAD_LIST || CC_TestParms)
4397            (COMMAND_DESCRIPTOR_t *)_TestParmsDataAddress,
4398    #endif // CC_TestParms
4399    #if (PAD_LIST || CC_Commit)
4400            (COMMAND_DESCRIPTOR_t *)_CommitDataAddress,
4401    #endif // CC_Commit
4402    #if (PAD_LIST || CC_PolicyPassword)
4403            (COMMAND_DESCRIPTOR_t *)_PolicyPasswordDataAddress,
4404    #endif // CC_PolicyPassword
```

```
4405    #if (PAD_LIST || CC_ZGen_2Phase)
4406            (COMMAND_DESCRIPTOR_t *)_ZGen_2PhaseDataAddress,
4407    #endif // CC_ZGen_2Phase
4408    #if (PAD_LIST || CC_EC_Ephemeral)
4409            (COMMAND_DESCRIPTOR_t *)_EC_EphemeralDataAddress,
4410    #endif // CC_EC_Ephemeral
4411    #if (PAD_LIST || CC_PolicyNvWritten)
4412            (COMMAND_DESCRIPTOR_t *)_PolicyNvWrittenDataAddress,
4413    #endif // CC_PolicyNvWritten
4414    #if (PAD_LIST || CC_PolicyTemplate)
4415            (COMMAND_DESCRIPTOR_t *)_PolicyTemplateDataAddress,
4416    #endif // CC_PolicyTemplate
4417    #if (PAD_LIST || CC_CreateLoaded)
4418            (COMMAND_DESCRIPTOR_t *)_CreateLoadedDataAddress,
4419    #endif // CC_CreateLoaded
4420    #if (PAD_LIST || CC_PolicyAuthorizeNV)
4421            (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeNVDataAddress,
4422    #endif // CC_PolicyAuthorizeNV
4423    #if (PAD_LIST || CC_EncryptDecrypt2)
4424            (COMMAND_DESCRIPTOR_t *)_EncryptDecrypt2DataAddress,
4425    #endif // CC_EncryptDecrypt2
4426    #if (PAD_LIST || CC_AC_GetCapability)
4427            (COMMAND_DESCRIPTOR_t *)_AC_GetCapabilityDataAddress,
4428    #endif // CC_AC_GetCapability
4429    #if (PAD_LIST || CC_AC_Send)
4430            (COMMAND_DESCRIPTOR_t *)_AC_SendDataAddress,
4431    #endif // CC_AC_Send
4432    #if (PAD_LIST || CC_Policy_AC_SendSelect)
4433            (COMMAND_DESCRIPTOR_t *)_Policy_AC_SendSelectDataAddress,
4434    #endif // CC_Policy_AC_SendSelect
4435    #if (PAD_LIST || CC_CertifyX509)
4436            (COMMAND_DESCRIPTOR_t *)_CertifyX509DataAddress,
4437    #endif // CC_CertifyX509
4438    #if (PAD_LIST || CC_ACT_SetTimeout)
4439            (COMMAND_DESCRIPTOR_t *)_ACT_SetTimeoutDataAddress,
4440    #endif // CC_ACT_SetTimeout
4441    #if (PAD_LIST || CC_Vendor_TCG_Test)
4442            (COMMAND_DESCRIPTOR_t *)_Vendor_TCG_TestDataAddress,
4443    #endif // CC_Vendor_TCG_Test
4444            0
4445    };
4446    #endif  // _COMMAND_TABLE_DISPATCH_
```

5.7 Commands.h

```
#ifndef _COMMANDS_H_
#define _COMMANDS_H_
```

Start-up

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

Testing

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

Session Commands

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

Object Commands

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

```
50  #endif
```

Duplication Commands

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

Asymmetric Primitives

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

Symmetric Primitives

```
78  #ifdef TPM_CC_EncryptDecrypt
79  #include "EncryptDecrypt_fp.h"
80  #endif
81  #ifdef TPM_CC_EncryptDecrypt2
82  #include "EncryptDecrypt2_fp.h"
83  #endif
84  #ifdef TPM_CC_Hash
85  #include "Hash_fp.h"
86  #endif
87  #ifdef TPM_CC_HMAC
88  #include "HMAC_fp.h"
89  #endif
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
135  #ifdef TPM_CC_CertifyX509
136  #include "CertifyX509_fp.h"
137  #endif
```

Ephemeral EC Keys

```
138  #ifdef TPM_CC_Commit
139  #include "Commit_fp.h"
140  #endif
141  #ifdef TPM_CC_EC_Ephemeral
142  #include "EC_Ephemeral_fp.h"
143  #endif
```

Signing and Signature Verification

```
144  #ifdef TPM_CC_VerifySignature
145  #include "VerifySignature_fp.h"
146  #endif
147  #ifdef TPM_CC_Sign
148  #include "Sign_fp.h"
149  #endif
```

Command Audit

```
150  #ifdef TPM_CC_SetCommandCodeAuditStatus
```

```
#include "SetCommandCodeAuditStatus_fp.h"
#endif
```

Integrity Collection (PCR)

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

Enhanced Authorization (EA) Commands

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

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

Hierarchy Commands

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

Dictionary Attack Functions

```
258   #ifdef TPM_CC_DictionaryAttackLockReset
259   #include "DictionaryAttackLockReset_fp.h"
260   #endif
261   #ifdef TPM_CC_DictionaryAttackParameters
262   #include "DictionaryAttackParameters_fp.h"
263   #endif
```

Miscellaneous Management Functions

```
264   #ifdef TPM_CC_PP_Commands
```

```
265 #include "PP_Commands_fp.h"
266 #endif
267 #ifdef TPM_CC_SetAlgorithmSet
268 #include "SetAlgorithmSet_fp.h"
269 #endif
```

Field Upgrade

```
270 #ifdef TPM_CC_FieldUpgradeStart
271 #include "FieldUpgradeStart_fp.h"
272 #endif
273 #ifdef TPM_CC_FieldUpgradeData
274 #include "FieldUpgradeData_fp.h"
275 #endif
276 #ifdef TPM_CC_FirmwareRead
277 #include "FirmwareRead_fp.h"
278 #endif
```

Context Management

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

Clocks and Timers

```
291 #ifdef TPM_CC_ReadClock
292 #include "ReadClock_fp.h"
293 #endif
294 #ifdef TPM_CC_ClockSet
295 #include "ClockSet_fp.h"
296 #endif
297 #ifdef TPM_CC_ClockRateAdjust
298 #include "ClockRateAdjust_fp.h"
299 #endif
```

Capability Commands

```
300 #ifdef TPM_CC_GetCapability
301 #include "GetCapability_fp.h"
302 #endif
303 #ifdef TPM_CC_TestParms
304 #include "TestParms_fp.h"
305 #endif
```

Non-volatile Storage

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

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

Attached Components

```
348   #ifdef TPM_CC_AC_GetCapability
349   #include "AC_GetCapability_fp.h"
350   #endif
351   #ifdef TPM_CC_AC_Send
352   #include "AC_Send_fp.h"
353   #endif
354   #ifdef TPM_CC_Policy_AC_SendSelect
355   #include "Policy_AC_SendSelect_fp.h"
356   #endif
```

Authenticated Countdown Timer

```
357   #ifdef TPM_CC_ACT_SetTimeout
358   #include "ACT_SetTimeout_fp.h"
359   #endif
```

Vendor Specific

```
360   #ifdef TPM_CC_Vendor_TCG_Test
361   #include "Vendor_TCG_Test_fp.h"
362   #endif
363   #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.

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

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

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

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

```
//#define _NO_CRT_STDIO_INLINE
```

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

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

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

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

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

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

Restore the compiler warning level

```
#   define _NORMAL_WARNING_LEVEL_                       \
__pragma(warning(pop))
#   include <stdint.h>
#endif
#ifndef _MSC_VER
#ifndef WINAPI
#   define WINAPI
#endif
#   define __pragma(x)
#   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  #include "Capabilities.h"
12  #include "TpmTypes.h"
13  #include "CommandAttributes.h"
14  #include "CryptTest.h"
15  #include "BnValues.h"
16  #include "CryptHash.h"
17  #include "CryptSym.h"
18  #include "CryptRand.h"
19  #include "CryptEcc.h"
20  #include "CryptRsa.h"
21  #include "CryptTest.h"
22  #include "TpmError.h"
23  #include "NV.h"
24  #include "ACT.h"
25
26  //** Defines and Types
27
28  //*** Size Types
29  // These types are used to differentiate the two different size values used.
30  //
31  // NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
32  typedef UINT16  NUMBYTES;
33
34  //*** Other Types
35  // An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
36  typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];
```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

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

A NAME is a BYTE array that can contain a TPMU_NAME

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

Definition for a PROOF value

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

Definition for a Primary Seed value

```
40  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.

```
41  #if CLOCK_STOPS
42  typedef UINT64          CLOCK_NONCE;
43  #else
44  typedef UINT32          CLOCK_NONCE;
45  #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.

```
46  typedef struct
47  {
48      unsigned            publicOnly : 1;     //0) SET if only the public portion of
49                                              //   an object is loaded
50      unsigned            epsHierarchy : 1;   //1) SET if the object belongs to EPS
51                                              //   Hierarchy
52      unsigned            ppsHierarchy : 1;   //2) SET if the object belongs to PPS
53                                              //   Hierarchy
54      unsigned            spsHierarchy : 1;   //3) SET f the object belongs to SPS
55                                              //   Hierarchy
56      unsigned            evict : 1;          //4) SET if the object is a platform or
57                                              //   owner evict object.  Platform-
58                                              //   evict object belongs to PPS
59                                              //   hierarchy, owner-evict object
60                                              //   belongs to SPS or EPS hierarchy.
61                                              //   This bit is also used to mark a
62                                              //   completed sequence object so it
63                                              //   will be flush when the
64                                              //   SequenceComplete command succeeds.
65      unsigned            primary : 1;        //5) SET for a primary object
66      unsigned            temporary : 1;      //6) SET for a temporary object
67      unsigned            stClear : 1;        //7) SET for an stClear object
68      unsigned            hmacSeq : 1;        //8) SET for an HMAC or MAC sequence
69                                              //   object
70      unsigned            hashSeq : 1;        //9) SET for a hash sequence object
71      unsigned            eventSeq : 1;       //10) SET for an event sequence object
```

```
72      unsigned            ticketSafe : 1;     //11) SET if a ticket is safe to create
73                                              //    for hash sequence object
74      unsigned            firstBlock : 1;     //12) SET if the first block of hash
75                                              //    data has been received.  It
76                                              //    works with ticketSafe bit
77      unsigned            isParent : 1;       //13) SET if the key has the proper
78                                              //    attributes to be a parent key
79  //   unsigned            privateExp : 1;     //14) SET when the private exponent
80  //                                          //    of an RSA key has been validated.
81      unsigned            not_used_14 : 1;
82      unsigned            occupied : 1;       //15) SET when the slot is occupied.
83      unsigned            derivation : 1;     //16) SET when the key is a derivation
84                                              //        parent
85      unsigned            external : 1;       //17) SET when the object is loaded with
86                                              //    TPM2_LoadExternal();
87  } OBJECT_ATTRIBUTES;
88  #if ALG_RSA
```

There is an overload of the sensitive.rsa.t.size field of a TPMT_SENSITIVE when an RSA key is loaded. When the sensitive->sensitive contains an RSA key with all of the CRT values, then the MSB of the size field will be set to indicate that the buffer contains all 5 of the CRT private key values.

```
89  #define     RSA_prime_flag      0x8000
90  #endif
```

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.

```
91  typedef struct OBJECT
92  {
93      // The attributes field is required to be first followed by the publicArea.
94      // This allows the overlay of the object structure and a sequence structure
95      OBJECT_ATTRIBUTES   attributes;         // object attributes
96      TPMT_PUBLIC         publicArea;         // public area of an object
97      TPMT_SENSITIVE      sensitive;          // sensitive area of an object
98      TPM2B_NAME          qualifiedName;      // object qualified name
99      TPMI_DH_OBJECT      evictHandle;        // if the object is an evict object,
100                                             // the original handle is kept here.
101                                             // The 'working' handle will be the
102                                             // handle of an object slot.
103     TPM2B_NAME          name;               // Name of the object name. Kept here
104                                             // to avoid repeatedly computing it.
105 } 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

```
106 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
```

```
142                                         //   This attribute is not SET unless
143                                         //   'isPolicy' is SET.
144     unsigned    isAuthValueNeeded : 1;  //5) SET if the authValue is required for
145                                         //   computing the session HMAC. This
146                                         //   attribute is not SET unless 'isPolicy'
147                                         //   is SET.
148     unsigned    isPasswordNeeded : 1;   //6) SET if a password authValue is required
149                                         //   for authorization This attribute is not
150                                         //   SET unless 'isPolicy' is SET.
151     unsigned    isPPRequired : 1;       //7) SET if physical presence is required to
152                                         //   be asserted when the authorization is
153                                         //   checked. This attribute is not SET
154                                         //   unless 'isPolicy' is SET.
155     unsigned    isTrialPolicy : 1;      //8) SET if the policy session is created
156                                         //   for trial of the policy's policyHash
157                                         //   generation. This attribute is not SET
158                                         //   unless 'isPolicy' is SET.
159     unsigned    isDaBound : 1;          //9) SET if the bind entity had noDA CLEAR.
160                                         //   If this is SET, then an authorization
161                                         //   failure using this session will count
162                                         //   against lockout even if the object
163                                         //   being authorized is exempt from DA.
164     unsigned    isLockoutBound : 1;     //10) SET if the session is bound to
165                                         //    lockoutAuth.
166     unsigned    includeAuth : 1;        //11) This attribute is SET when the
167                                         //    authValue of an object is to be
168                                         //    included in the computation of the
169                                         //    HMAC key for the command and response
170                                         //    computations. (was 'requestWasBound')
171     unsigned    checkNvWritten : 1;     //12) SET if the TPMA_NV_WRITTEN attribute
172                                         //    needs to be checked when the policy is
173                                         //    used for authorization for NV access.
174                                         //    If this is SET for any other type, the
175                                         //    policy will fail.
176     unsigned    nvWrittenState : 1;     //13) SET if TPMA_NV_WRITTEN is required to
177                                         //    be SET. Used when 'checkNvWritten' is
178                                         //    SET
179     unsigned    isTemplateSet : 1;      //14) SET if the templateHash needs to be
180                                         //    checked for Create, CreatePrimary, or
181                                         //    CreateLoaded.
182 } 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.

```
183 typedef struct SESSION
184 {
185     SESSION_ATTRIBUTES  attributes;         // session attributes
186     UINT32              pcrCounter;         // PCR counter value when PCR is
187                                             // included (policy session)
188                                             // If no PCR is included, this
189                                             // value is 0.
190     UINT64              startTime;          // The value in g_time when the session
191                                             // was started (policy session)
192     UINT64              timeout;            // The timeout relative to g_time
193                                             // There is no timeout if this value
194                                             // is 0.
195     CLOCK_NONCE         epoch;              // The g_clockEpoch value when the
196                                             // session was started. If g_clockEpoch
197                                             // does not match this value when the
198                                             // 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
```

```
252         //       to increment.
253         UINT32              pcrCounter;
254     } PCR_SAVE;
```

5.9.6.2 PCR_POLICY

```
255     #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.

```
256     typedef struct PCR_POLICY
257     {
258         TPMI_ALG_HASH       hashAlg[NUM_POLICY_PCR_GROUP];
259         TPM2B_DIGEST        a;
260         TPM2B_DIGEST        policy[NUM_POLICY_PCR_GROUP];
261     } PCR_POLICY;
262     #endif
```

5.9.6.3 PCR_AUTHVALUE

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

```
263     typedef struct PCR_AUTH_VALUE
264     {
265         TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
266     } 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().

```
267     typedef enum
268     {
269         SU_RESET,
270         SU_RESTART,
271         SU_RESUME
272     } 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.

```
273     typedef struct NV_INDEX
274     {
275         TPMS_NV_PUBLIC      publicArea;
276         TPM2B_AUTH          authValue;
277     } 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 or ROM as constant values.

5.9.10.2 Crypto Self-Test Values

```
EXTERN ALGORITHM_VECTOR     g_implementedAlgorithms;
EXTERN ALGORITHM_VECTOR     g_toTest;

//*** 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.
```

```
307  // This is an implementation choice and the same result can be achieved by using
308  // a macro.
309  #define g_rcIndexInitializer {  TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,            \
310                                  TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
     \
311                                  TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
     \
312                                  TPM_RC_D, TPM_RC_E, TPM_RC_F }
313  EXTERN const UINT16     g_rcIndex[15] INITIALIZER(g_rcIndexInitializer);
```

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.

```
314  EXTERN TPM_HANDLE       g_exclusiveAuditSession;
315
316  //*** g_time
317  // This is the value in which we keep the current command time. This is initialized
318  // at the start of each command. The time is the accumulated time since the last
319  // time that the TPM's timer was last powered up. Clock is the accumulated time
320  // since the last time that the TPM was cleared. g_time is in mS.
321  EXTERN  UINT64          g_time;
322
323  //*** g_timeEpoch
324  // This value contains the current clock Epoch. It changes when there is a clock
325  // discontinuity. It may be necessary to place this in NV should the timer be able
326  // to run across a power down of the TPM but not in all cases (e.g. dead battery).
327  // If the nonce is placed in NV, it should go in gp because it should be changing
328  // slowly.
329  #if CLOCK_STOPS
330  EXTERN CLOCK_NONCE       g_timeEpoch;
331  #else
332  #define g_timeEpoch      gp.timeEpoch
333  #endif
334
335
336  //*** g_phEnable
337  // This is the platform hierarchy control and determines if the platform hierarchy
338  // is available. This value is SET on each TPM2_Startup(). The default value is
339  // SET.
340  EXTERN BOOL             g_phEnable;
341
342  //*** g_pcrReConfig
343  // This value is SET if a TPM2_PCR_Allocate command successfully executed since
344  // the last TPM2_Startup(). If so, then the next shutdown is required to be
345  // Shutdown(CLEAR).
346  EXTERN BOOL             g_pcrReConfig;
347
348  //*** g_DRTMHandle
349  // This location indicates the sequence object handle that holds the DRTM
350  // sequence data. When not used, it is set to TPM_RH_UNASSIGNED. A sequence
351  // DRTM sequence is started on either _TPM_Init or _TPM_Hash_Start.
352  EXTERN TPMI_DH_OBJECT   g_DRTMHandle;
353
354  //*** g_DrtmPreStartup
355  // This value indicates that an H-CRTM occurred after _TPM_Init but before
356  // TPM2_Startup(). The define for PRE_STARTUP_FLAG is used to add the
357  // g_DrtmPreStartup value to gp_orderlyState at shutdown. This hack is to avoid
358  // adding another NV variable.
359  EXTERN  BOOL            g_DrtmPreStartup;
360
361  //*** g_StartupLocality3
362  // This value indicates that a TPM2_Startup() occurred at locality 3. Otherwise, it
```

```
363   // at locality 0. The define for STARTUP_LOCALITY_3 is to
364   // indicate that the startup was not at locality 0. This hack is to avoid
365   // adding another NV variable.
366   EXTERN  BOOL            g_StartupLocality3;
367
368   //***TPM_SU_NONE
369   // Part 2 defines the two shutdown/startup types that may be used in
370   // TPM2_Shutdown() and TPM2_Starup(). This additional define is
371   // used by the TPM to indicate that no shutdown was received.
372   // NOTE: This is a reserved value.
373   #define SU_NONE_VALUE           (0xFFFF)
374   #define TPM_SU_NONE             (TPM_SU)(SU_NONE_VALUE)
```

5.9.10.4 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.

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

5.9.10.5 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).

```
377   #define PRE_STARTUP_FLAG     0x8000
378   #define STARTUP_LOCALITY_3   0x4000
379   #if USE_DA_USED
```

5.9.10.6 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

```
380   EXTERN  BOOL                g_daUsed;
381   #endif
382
383   //*** g_updateNV
384   // This flag indicates if NV should be updated at the end of a command.
385   // This flag is set to UT_NONE at the beginning of each command in ExecuteCommand().
386   // This flag is checked in ExecuteCommand() after the detailed actions of a command
387   // complete. If the command execution was successful and this flag is not UT_NONE,
388   // any pending NV writes will be committed to NV.
389   // UT_ORDERLY causes any RAM data to be written to the orderly space for staging
390   // the write to NV.
391   typedef BYTE        UPDATE_TYPE;
392   #define UT_NONE     (UPDATE_TYPE)0
393   #define UT_NV       (UPDATE_TYPE)1
394   #define UT_ORDERLY  (UPDATE_TYPE)(UT_NV + 2)
395   EXTERN UPDATE_TYPE          g_updateNV;
396
397   //*** g_powerWasLost
398   // This flag is used to indicate if the power was lost. It is SET in _TPM__Init.
399   // This flag is cleared by TPM2_Startup() after all power-lost activities are
400   // completed.
```

```
401   // Note: When power is applied, this value can come up as anything. However,
402   // _plat__WasPowerLost() will provide the proper indication in that case. So, when
403   // power is actually lost, we get the correct answer. When power was not lost, but
404   // the power-lost processing has not been completed before the next _TPM_Init(),
405   // then the TPM still does the correct thing.
406   EXTERN BOOL             g_powerWasLost;
407
408   //*** g_clearOrderly
409   // This flag indicates if the execution of a command should cause the orderly
410   // state to be cleared.  This flag is set to FALSE at the beginning of each
411   // command in ExecuteCommand() and is checked in ExecuteCommand() after the
412   // detailed actions of a command complete but before the check of
413   // 'g_updateNV'. If this flag is TRUE, and the orderly state is not
414   // SU_NONE_VALUE, then the orderly state in NV memory will be changed to
415   // SU_NONE_VALUE or SU_DA_USED_VALUE.
416   EXTERN BOOL             g_clearOrderly;
417
418   //*** g_prevOrderlyState
419   // This location indicates how the TPM was shut down before the most recent
420   // TPM2_Startup(). This value, along with the startup type, determines if
421   // the TPM should do a TPM Reset, TPM Restart, or TPM Resume.
422   EXTERN TPM_SU           g_prevOrderlyState;
423
424   //*** g_nvOk
425   // This value indicates if the NV integrity check was successful or not. If not and
426   // the failure was severe, then the TPM would have been put into failure mode after
427   // it had been re-manufactured. If the NV failure was in the area where the state-save
428   // data is kept, then this variable will have a value of FALSE indicating that
429   // a TPM2_Startup(CLEAR) is required.
430   EXTERN BOOL             g_nvOk;
431   // NV availability is sampled as the start of each command and stored here
432   // so that its value remains consistent during the command execution
433   EXTERN TPM_RC           g_NvStatus;
434
435   //*** g_platformUnique
436   // This location contains the unique value(s) used to identify the TPM. It is
437   // loaded on every _TPM2_Startup()
438   // The first value is used to seed the RNG. The second value is used as a vendor
439   // authValue. The value used by the RNG would be the value derived from the
440   // chip unique value (such as fused) with a dependency on the authorities of the
441   // code in the TPM boot path. The second would be derived from the chip unique value
442   // with a dependency on the details of the code in the boot path. That is, the
443   // first value depends on the various signers of the code and the second depends on
444   // what was signed. The TPM vendor should not be able to know the first value but
445   // they are expected to know the second.
446   EXTERN TPM2B_AUTH       g_platformUniqueAuthorities; // Reserved for RNG
447
448   EXTERN TPM2B_AUTH       g_platformUniqueDetails;    // referenced by
      VENDOR_PERMANENT
449
450   //**********************************************************************************
451   //**********************************************************************************
452   //** Persistent Global Values
453   //**********************************************************************************
454   //**********************************************************************************
455   //*** Description
456   // The values in this section are global values that are persistent across power
457   // events. The lifetime of the values determines the structure in which the value
458   // is placed.
459
460   //**********************************************************************************
461   //*** PERSISTENT_DATA
462   //**********************************************************************************
463   // This structure holds the persistent values that only change as a consequence
464   // of a specific Protected Capability and are not affected by TPM power events
```

```
465  // (TPM2_Startup() or TPM2_Shutdown().
466  typedef struct
467  {
468  //*****************************************************************************
469  //          Hierarchy
470  //*****************************************************************************
471  // The values in this section are related to the hierarchies.
472
473      BOOL                disableClear;       // TRUE if TPM2_Clear() using
474                                              // lockoutAuth is disabled
475
476      // Hierarchy authPolicies
477      TPMI_ALG_HASH       ownerAlg;
478      TPMI_ALG_HASH       endorsementAlg;
479      TPMI_ALG_HASH       lockoutAlg;
480      TPM2B_DIGEST        ownerPolicy;
481      TPM2B_DIGEST        endorsementPolicy;
482      TPM2B_DIGEST        lockoutPolicy;
483
484      // Hierarchy authValues
485      TPM2B_AUTH          ownerAuth;
486      TPM2B_AUTH          endorsementAuth;
487      TPM2B_AUTH          lockoutAuth;
488
489      // Primary Seeds
490      TPM2B_SEED          EPSeed;
491      TPM2B_SEED          SPSeed;
492      TPM2B_SEED          PPSeed;
493      // Note there is a nullSeed in the state_reset memory.
494
495      // Hierarchy proofs
496      TPM2B_PROOF          phProof;
497      TPM2B_PROOF          shProof;
498      TPM2B_PROOF          ehProof;
499      // Note there is a nullProof in the state_reset memory.
500
501  //*****************************************************************************
502  //          Reset Events
503  //*****************************************************************************
504  // A count that increments at each TPM reset and never get reset during the life
505  // time of TPM.  The value of this counter is initialized to 1 during TPM
506  // manufacture process. It is used to invalidate all saved contexts after a TPM
507  // Reset.
508      UINT64              totalResetCount;
509
510  // This counter increments on each TPM Reset. The counter is reset by
511  // TPM2_Clear().
512      UINT32              resetCount;
513
514  //*****************************************************************************
515  //          PCR
516  //*****************************************************************************
517  // This structure hold the policies for those PCR that have an update policy.
518  // This implementation only supports a single group of PCR controlled by
519  // policy. If more are required, then this structure would be changed to
520  // an array.
521  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
522      PCR_POLICY          pcrPolicies;
523  #endif
524
525  // This structure indicates the allocation of PCR. The structure contains a
526  // list of PCR allocations for each implemented algorithm. If no PCR are
527  // allocated for an algorithm, a list entry still exists but the bit map
528  // will contain no SET bits.
529      TPML_PCR_SELECTION  pcrAllocated;
```

```
530
531 //*****************************************************************************
532 //          Physical Presence
533 //*****************************************************************************
534 // The PP_LIST type contains a bit map of the commands that require physical
535 // to be asserted when the authorization is evaluated. Physical presence will be
536 // checked if the corresponding bit in the array is SET and if the authorization
537 // handle is TPM_RH_PLATFORM.
538 //
539 // These bits may be changed with TPM2_PP_Commands().
540     BYTE                ppList[(COMMAND_COUNT + 7) / 8];
541
542 //*****************************************************************************
543 //          Dictionary attack values
544 //*****************************************************************************
545 // These values are used for dictionary attack tracking and control.
546     UINT32              failedTries;        // the current count of unexpired
547                                             // authorization failures
548
549     UINT32              maxTries;           // number of unexpired authorization
550                                             // failures before the TPM is in
551                                             // lockout
552
553     UINT32              recoveryTime;       // time between authorization failures
554                                             // before failedTries is decremented
555
556     UINT32              lockoutRecovery;    // time that must expire between
557                                             // authorization failures associated
558                                             // with lockoutAuth
559
560     BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
561                                             // allowed
562
563 //*****************************************************************************
564 //            Orderly State
565 //*****************************************************************************
566 // The orderly state for current cycle
567     TPM_SU              orderlyState;
568
569 //*****************************************************************************
570 //           Command audit values.
571 //*****************************************************************************
572     BYTE                auditCommands[((COMMAND_COUNT + 1) + 7) / 8];
573     TPMI_ALG_HASH       auditHashAlg;
574     UINT64              auditCounter;
575
576 //*****************************************************************************
577 //           Algorithm selection
578 //*****************************************************************************
579 //
580 // The 'algorithmSet' value indicates the collection of algorithms that are
581 // currently in used on the TPM.  The interpretation of value is vendor dependent.
582     UINT32              algorithmSet;
583
584 //*****************************************************************************
585 //           Firmware version
586 //*****************************************************************************
587 // The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is
588 // a scheme used in development to allow determination of the linker build time
589 // of the TPM. An actual implementation would implement these values in a way that
590 // is consistent with vendor needs. The values are maintained in RAM for simplified
591 // access with a master version in NV.  These values are modified in a
592 // vendor-specific way.
593
594 // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
595 // In the reference implementation, if this value is printed as a hex
```

```
596   // value, it will have the format of YYYYMMDD
597       UINT32              firmwareV1;
598
599   // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
600   // In the reference implementation, if this value is printed as a hex
601   // value, it will have the format of 00 HH MM SS
602       UINT32              firmwareV2;
603   //*****************************************************************************
604   //           Timer Epoch
605   //*****************************************************************************
606   // timeEpoch contains a nonce that has a vendor=specific size (should not be
607   // less than 8 bytes. This nonce changes when the clock epoch changes. The clock
608   // epoch changes when there is a discontinuity in the timing of the TPM.
609   #if !CLOCK_STOPS
610       CLOCK_NONCE         timeEpoch;
611   #endif
612
613   } PERSISTENT_DATA;
614   EXTERN PERSISTENT_DATA  gp;
615
616   //*****************************************************************************
617   //*****************************************************************************
618   //*** ORDERLY_DATA
619   //*****************************************************************************
620   //*****************************************************************************
621   // The data in this structure is saved to NV on each TPM2_Shutdown().
622   typedef struct orderly_data
623   {
624   //*****************************************************************************
625   //           TIME
626   //*****************************************************************************
627
628   // Clock has two parts. One is the state save part and one is the NV part. The
629   // state save version is updated on each command. When the clock rolls over, the
630   // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
631   // orderly way, then the sClock value is used to initialize the clock. If the
632   // TPM shutdown was not orderly, then the persistent value is used and the safe
633   // attribute is clear.
634
635       UINT64              clock;              // The orderly version of clock
636       TPMI_YES_NO         clockSafe;          // Indicates if the clock value is
637                                              // safe.
638
639       // In many implementations, the quality of the entropy available is not that
640       // high. To compensate, the current value of the drbgState can be saved and
641       // restored on each power cycle. This prevents the internal state from reverting
642       // to the initial state on each power cycle and starting with a limited amount
643       // of entropy. By keeping the old state and adding entropy, the entropy will
644       // accumulate.
645       DRBG_STATE          drbgState;
646
647   // These values allow the accumulation of self-healing time across orderly shutdown
648   // of the TPM.
649   #if ACCUMULATE_SELF_HEAL_TIMER
650       UINT64              selfHealTimer;  // current value of s_selfHealTimer
651       UINT64              lockoutTimer;   // current value of s_lockoutTimer
652       UINT64              time;           // current value of g_time at shutdown
653   #endif // ACCUMULATE_SELF_HEAL_TIMER
654
655   // These are the ACT Timeout values. They are saved with the other timers
656   #define DefineActData(N)  ACT_STATE      ACT_##N;
657       FOR_EACH_ACT(DefineActData)
658
659    // this is the 'signaled' attribute data for all the ACT. It is done this way so
660    // that they can be manipulated by ACT number rather than having to access a
```

```
661     // structure.
662        UINT32              signaledACT;
663     } ORDERLY_DATA;
664     #if ACCUMULATE_SELF_HEAL_TIMER
665     #define     s_selfHealTimer     go.selfHealTimer
666     #define     s_lockoutTimer      go.lockoutTimer
667     #endif  // ACCUMULATE_SELF_HEAL_TIMER
668     #  define drbgDefault go.drbgState
669     EXTERN ORDERLY_DATA     go;
670
671     //*****************************************************************************
672     //*****************************************************************************
673     //*** STATE_CLEAR_DATA
674     //*****************************************************************************
675     //*****************************************************************************
676     // This structure contains the data that is saved on Shutdown(STATE)
677     // and restored on Startup(STATE).  The values are set to their default
678     // settings on any Startup(Clear). In other words, the data is only persistent
679     // across TPM Resume.
680     //
681     // If the comments associated with a parameter indicate a default reset value, the
682     // value is applied on each Startup(CLEAR).
683
684     typedef struct state_clear_data
685     {
686     //*****************************************************************************
687     //           Hierarchy Control
688     //*****************************************************************************
689         BOOL                shEnable;           // default reset is SET
690         BOOL                ehEnable;           // default reset is SET
691         BOOL                phEnableNV;         // default reset is SET
692         TPMI_ALG_HASH       platformAlg;        // default reset is TPM_ALG_NULL
693         TPM2B_DIGEST        platformPolicy;     // default reset is an Empty Buffer
694         TPM2B_AUTH          platformAuth;       // default reset is an Empty Buffer
695
696     //*****************************************************************************
697     //           PCR
698     //*****************************************************************************
699     // The set of PCR to be saved on Shutdown(STATE)
700         PCR_SAVE            pcrSave;            // default reset is 0...0
701
702     // This structure hold the authorization values for those PCR that have an
703     // update authorization.
704     // This implementation only supports a single group of PCR controlled by
705     // authorization. If more are required, then this structure would be changed to
706     // an array.
707         PCR_AUTHVALUE       pcrAuthValues;
708
709     //*****************************************************************************
710     //           ACT
711     //*****************************************************************************
712     #define DefineActPolicySpace(N)     TPMT_HA     act_##N;
713         FOR_EACH_ACT(DefineActPolicySpace)
714
715     } STATE_CLEAR_DATA;
716     EXTERN STATE_CLEAR_DATA gc;
717
718     //*****************************************************************************
719     //*****************************************************************************
720     //***  State Reset Data
721     //*****************************************************************************
722     //*****************************************************************************
723     // This structure contains data is that is saved on Shutdown(STATE) and restored on
724     // the subsequent Startup(ANY). That is, the data is preserved across TPM Resume
725     // and TPM Restart.
```

```
726 //
727 // If a default value is specified in the comments this value is applied on
728 // TPM Reset.
729
730 typedef struct state_reset_data
731 {
732 //*****************************************************************************
733 //          Hierarchy Control
734 //*****************************************************************************
735     TPM2B_PROOF         nullProof;          // The proof value associated with
736                                             // the TPM_RH_NULL hierarchy. The
737                                             // default reset value is from the RNG.
738
739     TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
740                                             // hierarchy. The default reset value
741                                             // is from the RNG.
742
743 //*****************************************************************************
744 //           Context
745 //*****************************************************************************
746 // The 'clearCount' counter is incremented each time the TPM successfully executes
747 // a TPM Resume. The counter is included in each saved context that has 'stClear'
748 // SET (including descendants of keys that have 'stClear' SET). This prevents these
749 // objects from being loaded after a TPM Resume.
750 // If 'clearCount' is at its maximum value when the TPM receives a Shutdown(STATE),
751 // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
752     UINT32              clearCount;         // The default reset value is 0.
753
754     UINT64              objectContextID;    // This is the context ID for a saved
755                                             //  object context. The default reset
756                                             //  value is 0.
757     CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];    // This array
    contains
758                                             // contains the values used to track
759                                             // the version numbers of saved
760                                             // contexts (see
761                                             // Session.c in for details). The
762                                             // default reset value is {0}.
763
764     CONTEXT_COUNTER     contextCounter;     // This is the value from which the
765                                             // 'contextID' is derived. The
766                                             // default reset value is {0}.
767
768 //*****************************************************************************
769 //           Command Audit
770 //*****************************************************************************
771 // When an audited command completes, ExecuteCommand() checks the return
772 // value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
773 // TPM will extend the cpHash and rpHash for the command to this value. If this
774 // digest was the Zero Digest before the cpHash was extended, the audit counter
775 // is incremented.
776
777     TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
778                                             // by TPM2_GetCommandAuditDigest() or a
779                                             // TPM Reset.
780
781 //*****************************************************************************
782 //           Boot counter
783 //*****************************************************************************
784
785     UINT32              restartCount;       // This counter counts TPM Restarts.
786                                             // The default reset value is 0.
787
788 //*****************************************************************************
789 //            PCR
```

```
790   //*****************************************************************************
791   // This counter increments whenever the PCR are updated. This counter is preserved
792   // across TPM Resume even though the PCR are not preserved. This is because
793   // sessions remain active across TPM Restart and the count value in the session
794   // is compared to this counter so this counter must have values that are unique
795   // as long as the sessions are active.
796   // NOTE: A platform-specific specification may designate that certain PCR changes
797   //       do not increment this counter to increment.
798       UINT32              pcrCounter;         // The default reset value is 0.
799
800   #if     ALG_ECC
801
802   //*****************************************************************************
803   //         ECDAA
804   //*****************************************************************************
805       UINT64              commitCounter;      // This counter increments each time
806                                               // TPM2_Commit() returns
807                                               // TPM_RC_SUCCESS. The default reset
808                                               // value is 0.
809
810       TPM2B_NONCE         commitNonce;        // This random value is used to compute
811                                               // the commit values. The default reset
812                                               // value is from the RNG.
813
814   // This implementation relies on the number of bits in g_commitArray being a
815   // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
816       BYTE                 commitArray[16];   // The default reset value is {0}.
817
818   #endif // ALG_ECC
819   } STATE_RESET_DATA;
820
821   EXTERN STATE_RESET_DATA gr;
822
823   //** NV Layout
824   // The NV data organization is
825   // 1) a PERSISTENT_DATA structure
826   // 2) a STATE_RESET_DATA structure
827   // 3) a STATE_CLEAR_DATA structure
828   // 4) an ORDERLY_DATA structure
829   // 5) the user defined NV index space
830   #define NV_PERSISTENT_DATA  (0)
831   #define NV_STATE_RESET_DATA (NV_PERSISTENT_DATA + sizeof(PERSISTENT_DATA))
832   #define NV_STATE_CLEAR_DATA (NV_STATE_RESET_DATA + sizeof(STATE_RESET_DATA))
833   #define NV_ORDERLY_DATA     (NV_STATE_CLEAR_DATA + sizeof(STATE_CLEAR_DATA))
834   #define NV_INDEX_RAM_DATA   (NV_ORDERLY_DATA + sizeof(ORDERLY_DATA))
835   #define NV_USER_DYNAMIC     (NV_INDEX_RAM_DATA + sizeof(s_indexOrderlyRam))
836   #define NV_USER_DYNAMIC_END     NV_MEMORY_SIZE
```

5.9.11 Global Macro Definitions

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

```
837   #define NV_READ_PERSISTENT(to, from)                \
838               NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
839   #define NV_WRITE_PERSISTENT(to, from)               \
840               NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
841   #define CLEAR_PERSISTENT(item)                      \
842               NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
843   #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().

```
844  typedef UINT16      COMMAND_INDEX;
845  #define UNIMPLEMENTED_COMMAND_INDEX     ((COMMAND_INDEX)(~0))
846  typedef struct _COMMAND_FLAGS_
847  {
848      unsigned    trialPolicy : 1;    //1) If SET, one of the handles references a
849                                      //   trial policy and authorization may be
850                                      //   skipped. This is only allowed for a policy
851                                      //   command.
852  } 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.

```
853  typedef struct _COMMAND_
854  {
855      TPM_ST           tag;               // the parsed command tag
856      TPM_CC           code;              // the parsed command code
857      COMMAND_INDEX    index;             // the computed command index
858      UINT32           handleNum;         // the number of entity handles in the
859                                          //   handle area of the command
860      TPM_HANDLE       handles[MAX_HANDLE_NUM]; // the parsed handle values
861      UINT32           sessionNum;        // the number of sessions found
862      INT32            parameterSize;     // starts out with the parsed command size
863                                          // and is reduced and values are
864                                          // unmarshaled. Just before calling the
865                                          // command actions, this should be zero.
866                                          // After the command actions, this number
867                                          // should grow as values are marshaled
868                                          // in to the response buffer.
869      INT32            authSize;          // this is initialized with the parsed size
870                                          // of authorizationSize field and should
871                                          // be zero when the authorizations are
872                                          // parsed.
873      BYTE            *parameterBuffer;   // input to ExecuteCommand
874      BYTE            *responseBuffer;    // input to ExecuteCommand
875  #if ALG_SHA1
876      TPM2B_SHA1_DIGEST   sha1CpHash;
877      TPM2B_SHA1_DIGEST   sha1RpHash;
878  #endif
879  #if ALG_SHA256
880      TPM2B_SHA256_DIGEST sha256CpHash;
881      TPM2B_SHA256_DIGEST sha256RpHash;
882  #endif
883  #if ALG_SHA384
884      TPM2B_SHA384_DIGEST sha384CpHash;
885      TPM2B_SHA384_DIGEST sha384RpHash;
886  #endif
887  #if ALG_SHA512
888      TPM2B_SHA512_DIGEST sha512CpHash;
889      TPM2B_SHA512_DIGEST sha512RpHash;
890  #endif
891  #if ALG_SM3_256
892      TPM2B_SM3_256_DIGEST sm3_256CpHash;
893      TPM2B_SM3_256_DIGEST sm3_256RpHash;
894  #endif
895  } COMMAND;
```

Global sting constants for consistency in KDF function calls. These string constants are shared across functions to make sure that they are all using consistent sting values.

```
896  #define STRING_INITIALIZER(value)   {{sizeof(value), {value}}}
897  #define TPM2B_STRING(name, value)                                          \
898  typedef union name##_ {                                                    \
899          struct  {                                                          \
900              UINT16  size;                                                  \
901              BYTE    buffer[sizeof(value)];                                 \
902          } t;                                                               \
903          TPM2B   b;                                                         \
904      } TPM2B_##name##_;                                                     \
905  EXTERN  const TPM2B_##name##_      name##_ INITIALIZER(STRING_INITIALIZER(value));  \
906  EXTERN  const TPM2B               *name INITIALIZER(&name##_.b)
907  TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
908  TPM2B_STRING(CFB_KEY, "CFB");
909  TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
910  TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
911  TPM2B_STRING(SECRET_KEY, "SECRET");
912  TPM2B_STRING(SESSION_KEY, "ATH");
913  TPM2B_STRING(STORAGE_KEY, "STORAGE");
914  TPM2B_STRING(XOR_KEY, "XOR");
915  TPM2B_STRING(COMMIT_STRING, "ECDAA Commit");
916  TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
917  TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
918  TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
919  #if SELF_TEST
920  TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
921  #endif // SELF_TEST
```

5.9.12 From CryptTest.c

This structure contains the self-test state values for the cryptographic modules.

```
922  EXTERN CRYPTO_SELF_TEST_STATE   g_cryptoSelfTestState;
923
924  //**********************************************************************
925  //** From Manufacture.c
926  //**********************************************************************
927  EXTERN BOOL              g_manufactured INITIALIZER(FALSE);
```

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.

```
928  EXTERN BOOL              g_initialized;
929
930  //** Private data
931
932  //**********************************************************************
933  //*** From SessionProcess.c
934  //**********************************************************************
935  #if defined SESSION_PROCESS_C || defined GLOBAL_C || defined MANUFACTURE_C
936  // The following arrays are used to save command sessions information so that the
937  // command handle/session buffer does not have to be preserved for the duration of
938  // the command. These arrays are indexed by the session index in accordance with
939  // the order of sessions in the session area of the command.
940  //
941  // Array of the authorization session handles
942  EXTERN TPM_HANDLE       s_sessionHandles[MAX_SESSION_NUM];
943
944  // Array of authorization session attributes
945  EXTERN TPMA_SESSION     s_attributes[MAX_SESSION_NUM];
946
947  // Array of handles authorized by the corresponding authorization sessions;
948  // and if none, then TPM_RH_UNASSIGNED value is used
```

```
949  EXTERN TPM_HANDLE       s_associatedHandles[MAX_SESSION_NUM];
950
951  // Array of nonces provided by the caller for the corresponding sessions
952  EXTERN TPM2B_NONCE      s_nonceCaller[MAX_SESSION_NUM];
953
954  // Array of authorization values (HMAC's or passwords) for the corresponding
955  // sessions
956  EXTERN TPM2B_AUTH       s_inputAuthValues[MAX_SESSION_NUM];
957
958  // Array of pointers to the SESSION structures for the sessions in a command
959  EXTERN SESSION          *s_usedSessions[MAX_SESSION_NUM];
960
961  // Special value to indicate an undefined session index
962  #define             UNDEFINED_INDEX     (0xFFFF)
```

Index of the session used for encryption of a response parameter

```
963  EXTERN UINT32           s_encryptSessionIndex;
964
965  // Index of the session used for decryption of a command parameter
966  EXTERN UINT32           s_decryptSessionIndex;
967
968  // Index of a session used for audit
969  EXTERN UINT32           s_auditSessionIndex;
970
971  // The cpHash for command audit
972  #ifdef  TPM_CC_GetCommandAuditDigest
973  EXTERN TPM2B_DIGEST    s_cpHashForCommandAudit;
974  #endif
975
976  // Flag indicating if NV update is pending for the lockOutAuthEnabled or
977  // failedTries DA parameter
978  EXTERN BOOL             s_DAPendingOnNV;
979
980  #endif // SESSION_PROCESS_C
981
982  //**********************************************************************
983  //*** From DA.c
984  //**********************************************************************
985  #if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
986  // This variable holds the accumulated time since the last time
987  // that 'failedTries' was decremented. This value is in millisecond.
988  #if !ACCUMULATE_SELF_HEAL_TIMER
989  EXTERN UINT64       s_selfHealTimer;
990
991  // This variable holds the accumulated time that the lockoutAuth has been
992  // blocked.
993  EXTERN UINT64       s_lockoutTimer;
994  #endif // ACCUMULATE_SELF_HEAL_TIMER
995
996  #endif // DA_C
997
998  //**********************************************************************
999  //*** From NV.c
1000 //**********************************************************************
1001 #if defined NV_C || defined GLOBAL_C
1002 // This marks the end of the NV area. This is a run-time variable as it might
1003 // not be compile-time constant.
1004 EXTERN NV_REF   s_evictNvEnd;
1005
1006 // This space is used to hold the index data for an orderly Index. It also contains
1007 // the attributes for the index.
1008 EXTERN BYTE      s_indexOrderlyRam[RAM_INDEX_SPACE];   // The orderly NV Index data
1009
1010 // This value contains the current max counter value. It is written to the end of
```

```
1011    // allocatable NV space each time an index is deleted or added. This value is
1012    // initialized on Startup. The indices are searched and the maximum of all the
1013    // current counter indices and this value is the initial value for this.
1014    EXTERN UINT64    s_maxCounter;
1015
1016    // This is space used for the NV Index cache. As with a persistent object, the
1017    // contents of a referenced index are copied into the cache so that the
1018    // NV Index memory scanning and data copying can be reduced.
1019    // Only code that operates on NV Index data should use this cache directly. When
1020    // that action code runs, s_lastNvIndex will contain the index header information.
1021    // It will have been loaded when the handles were verified.
1022    // NOTE: An NV index handle can appear in many commands that do not operate on the
1023    // NV data (e.g. TPM2_StartAuthSession). However, only one NV Index at a time is
1024    // ever directly referenced by any command. If that changes, then the NV Index
1025    // caching needs to be changed to accommodate that. Currently, the code will verify
1026    // that only one NV Index is referenced by the handles of the command.
1027    EXTERN      NV_INDEX         s_cachedNvIndex;
1028    EXTERN      NV_REF           s_cachedNvRef;
1029    EXTERN      BYTE            *s_cachedNvRamRef;
1030
1031    // Initial NV Index/evict object iterator value
1032    #define     NV_REF_INIT     (NV_REF)0xFFFFFFFF
1033    #endif
```

5.9.12.1 From Object.c

```
1034    #if defined OBJECT_C || defined GLOBAL_C
```

This type is the container for an object.

```
1035    EXTERN OBJECT           s_objects[MAX_LOADED_OBJECTS];
1036
1037    #endif // OBJECT_C
1038
1039    //**********************************************************************
1040    //*** From PCR.c
1041    //**********************************************************************
1042    #if defined PCR_C || defined GLOBAL_C
1043    typedef struct
1044    {
1045    #if     ALG_SHA1
1046        // SHA1 PCR
1047        BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
1048    #endif
1049    #if     ALG_SHA256
1050        // SHA256 PCR
1051        BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
1052    #endif
1053    #if     ALG_SHA384
1054        // SHA384 PCR
1055        BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
1056    #endif
1057    #if     ALG_SHA512
1058        // SHA512 PCR
1059        BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
1060    #endif
1061    #if     ALG_SM3_256
1062        // SHA256 PCR
1063        BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
1064    #endif
1065    } PCR;
1066
1067    typedef struct
```

```
1068    {
1069        unsigned int    stateSave : 1;              // if the PCR value should be
1070                                                    // saved in state save
1071        unsigned int    resetLocality : 5;          // The locality that the PCR
1072                                                    // can be reset
1073        unsigned int    extendLocality : 5;         // The locality that the PCR
1074                                                    // can be extend
1075    } PCR_Attributes;
1076
1077    EXTERN PCR          s_pcrs[IMPLEMENTATION_PCR];
1078
1079    #endif // PCR_C
1080
1081    //**********************************************************************
1082    //*** From Session.c
1083    //**********************************************************************
1084    #if defined SESSION_C || defined GLOBAL_C
1085    // Container for HMAC or policy session tracking information
1086    typedef struct
1087    {
1088        BOOL                occupied;
1089        SESSION             session;        // session structure
1090    } SESSION_SLOT;
1091
1092    EXTERN SESSION_SLOT     s_sessions[MAX_LOADED_SESSIONS];
1093
1094    //  The index in contextArray that has the value of the oldest saved session
1095    //  context. When no context is saved, this will have a value that is greater
1096    //  than or equal to MAX_ACTIVE_SESSIONS.
1097    EXTERN UINT32            s_oldestSavedSession;
1098
1099    // The number of available session slot openings.  When this is 1,
1100    // a session can't be created or loaded if the GAP is maxed out.
1101    // The exception is that the oldest saved session context can always
1102    // be loaded (assuming that there is a space in memory to put it)
1103    EXTERN int               s_freeSessionSlots;
1104
1105    #endif // SESSION_C
1106
1107    //**********************************************************************
1108    //*** From IoBuffers.c
1109    //**********************************************************************
1110    #if defined IO_BUFFER_C || defined GLOBAL_C
1111    // Each command function is allowed a structure for the inputs to the function and
1112    // a structure for the outputs. The command dispatch code unmarshals the input butter
1113    // to the command action input structure starting at the first byte of
1114    // s_actionIoBuffer. The value of s_actionIoAllocation is the number of UINT64 values
1115    // allocated. It is used to set the pointer for the response structure. The command
1116    // dispatch code will marshal the response values into the final output buffer.
1117    EXTERN UINT64   s_actionIoBuffer[768];      // action I/O buffer
1118    EXTERN UINT32   s_actionIoAllocation;       // number of UIN64 allocated for the
1119                                                // action input structure
1120    #endif // IO_BUFFER_C
1121
1122    //**********************************************************************
1123    //*** From TPMFail.c
1124    //**********************************************************************
1125    // This value holds the address of the string containing the name of the function
1126    // in which the failure occurred. This address value isn't useful for anything
1127    // other than helping the vendor to know in which file the failure  occurred.
1128    EXTERN BOOL      g_inFailureMode;       // Indicates that the TPM is in failure mode
1129    #if SIMULATION
1130    EXTERN BOOL      g_forceFailureMode;    // flag to force failure mode during test
1131    #endif
1132
```

```
1133    typedef void(FailFunction)(const char *function, int line, int code);
1134    #if defined TPM_FAIL_C || defined GLOBAL_C
1135    EXTERN UINT32    s_failFunction;
1136    EXTERN UINT32    s_failLine;           // the line in the file at which
1137                                           // the error was signaled
1138    EXTERN UINT32    s_failCode;           // the error code used
1139
1140    EXTERN FailFunction    *LibFailCallback;
1141
1142    #endif // TPM_FAIL_C
1143
1144    //**********************************************************************
1145    //*** From ACT_spt.c
1146    //**********************************************************************
1147    // This value is used to indicate if an ACT has been updated since the last
1148    // TPM2_Startup() (one bit for each ACT). If the ACT is not updated
1149    // (TPM2_ACT_SetTimeout()) after a startup, then on each TPM2_Shutdown() the TPM will
1150    // save 1/2 of the current timer value. This prevents an attack on the ACT by saving
1151    // the counter and then running for a long period of time before doing a TPM Restart.
1152    // A quick TPM2_Shutdown() after each
1153    EXTERN UINT16                        s_ActUpdated;
1154
1155    //**********************************************************************
1156    //*** From CommandCodeAttributes.c
1157    //**********************************************************************
1158    // This array is instanced in CommandCodeAttributes.c when it includes
1159    // CommandCodeAttributes.h. Don't change the extern to EXTERN.
1160    extern  const  TPMA_CC               s_ccAttr[];
1161    extern  const  COMMAND_ATTRIBUTES    s_commandAttributes[];
1162
1163    #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))
#   define LOG_FAILURE(errorCode) (TpmLogFailure(errorCode))
#else
#   define FAIL(errorCode)        TpmFail(FUNCTION_NAME, __LINE__, errorCode)
#   define LOG_FAILURE(errorCode) TpmLogFailure(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.

```
37  #define REQUIRE(condition, errorCode, returnCode)                                \
38      {                                                                            \
39          if(!!(condition))                                                        \
40          {                                                                        \
41              FAIL(FATAL_ERROR_errorCode);                                         \
42              FAIL_RETURN(returnCode);                                             \
43          }                                                                        \
44      }
45  #define PARAMETER_CHECK(condition, returnCode)                                   \
46      REQUIRE((condition), PARAMETER, returnCode)
47  #if (defined EMPTY_ASSERT) && (EMPTY_ASSERT != NO)
48  #   define pAssert(a)  ((void)0)
49  #else
50  #   define pAssert(a) {if(!(a)) FAIL(FATAL_ERROR_PARAMETER);}
51  #endif
```

5.10.4 Derived from Vendor-specific values

Values derived from vendor specific settings in TpmProfile.h

```
52  #define PCR_SELECT_MIN          ((PLATFORM_PCR+7)/8)
53  #define PCR_SELECT_MAX          ((IMPLEMENTATION_PCR+7)/8)
54  #define MAX_ORDERLY_COUNT       ((1 << ORDERLY_BITS) - 1)
55  #define RSA_MAX_PRIME           (MAX_RSA_KEY_BYTES / 2)
56  #define RSA_PRIVATE_SIZE        (RSA_MAX_PRIME * 5)
```

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

```
57  #if COMPILER_CHECKS
58  #   define  cAssert     pAssert
59  #else
60  #   define cAssert(value)
61  #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.

```
62  #define     ERROR_RETURN(returnCode)                                             \
63      {                                                                            \
64           retVal = returnCode;                                                    \
65           goto Exit;                                                              \
66      }
67  #ifndef MAX
68  #  define MAX(a, b) ((a) > (b) ? (a) : (b))
69  #endif
70  #ifndef MIN
71  #  define MIN(a, b) ((a) < (b) ? (a) : (b))
72  #endif
73  #ifndef IsOdd
```

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

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

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

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

```
#ifndef LIB_EXPORT
#   define LIB_EXPORT
#   define LIB_IMPORT
#endif
#ifndef NORETURN
#   define NORETURN _Noreturn
#endif
#ifndef NOT_REFERENCED
#   define NOT_REFERENCED(x = x)    ((void) (x))
#endif
#define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
#define JOIN(x, y) x##y
#define JOIN3(x, y, z) x##y##z
#define CONCAT(x, y) JOIN(x, y)
#define CONCAT3(x, y, z) JOIN3(x,y,z)
```

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.

```
#ifndef CONTEXT_HASH_ALGORITHM
#   if defined ALG_SHA512 && ALG_SHA512 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA512
#   elif defined ALG_SHA384 && ALG_SHA384 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA384
#   elif defined ALG_SHA256 && ALG_SHA256 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA256
#   elif defined ALG_SM3_256 && ALG_SM3_256 == YES
#       define CONTEXT_HASH_ALGORITHM    SM3_256
#   elif defined ALG_SHA1 && ALG_SHA1 == YES
#       define CONTEXT_HASH_ALGORITHM  SHA1
#   endif
#   define CONTEXT_INTEGRITY_HASH_ALG  CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
#endif
#ifndef CONTEXT_INTEGRITY_HASH_SIZE
#define CONTEXT_INTEGRITY_HASH_SIZE CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
#endif
#if     ALG_RSA
#define     RSA_SECURITY_STRENGTH (MAX_RSA_KEY_BITS >= 15360 ? 256 :                \
                                   (MAX_RSA_KEY_BITS >=  7680 ? 192 :               \
                                   (MAX_RSA_KEY_BITS >=  3072 ? 128 :               \
                                   (MAX_RSA_KEY_BITS >=  2048 ? 112 :               \
                                   (MAX_RSA_KEY_BITS >=  1024 ?  80 :  0)))))
#else
#define     RSA_SECURITY_STRENGTH   0
#endif // ALG_RSA
#if     ALG_ECC
#define     ECC_SECURITY_STRENGTH (MAX_ECC_KEY_BITS >= 521 ? 256 :                  \
                                   (MAX_ECC_KEY_BITS >= 384 ? 192 :                 \
                                   (MAX_ECC_KEY_BITS >= 256 ? 128 : 0)))
```

```
127 #else
128 #define     ECC_SECURITY_STRENGTH   0
129 #endif // ALG_ECC
130 #define     MAX_ASYM_SECURITY_STRENGTH                                          \
131                         MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
132 #define     MAX_HASH_SECURITY_STRENGTH  ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)
```

Unless some algorithm is broken...

```
133 #define     MAX_SYM_SECURITY_STRENGTH   MAX_SYM_KEY_BITS
134 #define MAX_SECURITY_STRENGTH_BITS                                              \
135                         MAX(MAX_ASYM_SECURITY_STRENGTH,                         \
136                         MAX(MAX_SYM_SECURITY_STRENGTH,                          \
137                             MAX_HASH_SECURITY_STRENGTH))
```

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

```
138 #define PROOF_SIZE      CONTEXT_INTEGRITY_HASH_SIZE
```

As required by P1.14.4

```
139 #define COMPLIANT_PROOF_SIZE                                                    \
140             (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))
```

As required by P1.14.3.1

```
141 #define COMPLIANT_PRIMARY_SEED_SIZE                                             \
142     BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)
```

This is the pre-errata version

```
143 #ifndef PRIMARY_SEED_SIZE
144 #   define PRIMARY_SEED_SIZE    PROOF_SIZE
145 #endif
146 #if USE_SPEC_COMPLIANT_PROOFS
147 #   undef PROOF_SIZE
148 #   define PROOF_SIZE           COMPLIANT_PROOF_SIZE
149 #   undef PRIMARY_SEED_SIZE
150 #   define PRIMARY_SEED_SIZE    COMPLIANT_PRIMARY_SEED_SIZE
151 #endif  // USE_SPEC_COMPLIANT_PROOFS
152 #if !SKIP_PROOF_ERRORS
153 #   if PROOF_SIZE < COMPLIANT_PROOF_SIZE
154 #       error "PROOF_SIZE is not compliant with TPM specification"
155 #   endif
156 #   if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
157 #       error Non-compliant PRIMARY_SEED_SIZE
158 #   endif
159 #endif // !SKIP_PROOF_ERRORS
```

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

```
160 #if defined CONTEXT_ENCRYPT_ALG
161 #   undef CONTEXT_ENCRYPT_ALGORITHM
162 #   if   CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
163 #       define CONTEXT_ENCRYPT_ALGORITHM    AES
164 #   elif CONTEXT_ENCRYPT_ALG == ALG_SM4_VALUE
165 #       define CONTEXT_ENCRYPT_ALGORITHM    SM4
166 #   elif CONTEXT_ENCRYPT_ALG == ALG_CAMELLIA_VALUE
167 #       define CONTEXT_ENCRYPT_ALGORITHM    CAMELLIA
168 #   elif CONTEXT_ENCRYPT_ALG == ALG_TDES_VALUE
169 #   error Are you kidding?
170 #   else
171 #       error Unknown value for CONTEXT_ENCRYPT_ALG
```

```
#   endif // CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
#else
#   define CONTEXT_ENCRYPT_ALG                                                 \
            CONCAT3(ALG_, CONTEXT_ENCRYPT_ALGORITHM, _VALUE)
#endif  // CONTEXT_ENCRYPT_ALG
#define CONTEXT_ENCRYPT_KEY_BITS                                               \
                CONCAT(CONTEXT_ENCRYPT_ALGORITHM, _MAX_KEY_SIZE_BITS)
#define CONTEXT_ENCRYPT_KEY_BYTES       ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
```

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

```
#ifndef LABEL_MAX_BUFFER
#define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
#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. Stealing 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)
#   define TPMA_ZERO_INITIALIZER()          {0}
#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)
#   define TPMA_ZERO_INITIALIZER()          (0)
#endif
#define VERIFY(_X) if(!(_X)) goto Error
```

These macros determine if the values in this file are referenced or instanced. Global.c defines GLOBAL_C so all the values in this file will be instanced in Global.obj. For all other files that include this file, the values will simply be external references. For constants, there can be an initializer.

```
#ifdef GLOBAL_C
#define EXTERN
#define INITIALIZER(_value_)  = _value_
#else
#define EXTERN  extern
#define INITIALIZER(_value_)
#endif
```

This macro will create an OID. All OIDs are in DER form with a first octet of 0x06 indicating an OID fallowed by an octet indicating the number of octets in the rest of the OID. This allows a user of this OID to know how much/little to copy.

```
209  #define MAKE_OID(NAME)                                                    \
210          EXTERN  const BYTE OID##NAME[] INITIALIZER({OID##NAME##_VALUE})
```

This definition is moved from TpmProfile.h because it is not actually vendor- specific. It has to be the same size as the *sequence* parameter of a TPMS_CONTEXT and that is a UINT64. So, this is an invariant value

```
211  #define CONTEXT_COUNTER         UINT64
212  #endif // GP_MACROS_H
```


5.11 InternalRoutines.h

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

DRTM functions

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

Internal subsystem functions

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

Internal support functions

```
#include "CommandCodeAttributes_fp.h"
#include "Marshal.h"
#include "Time_fp.h"
#include "Locality_fp.h"
#include "PP_fp.h"
#include "CommandAudit_fp.h"
#include "Manufacture_fp.h"
#include "Handle_fp.h"
#include "Power_fp.h"
#include "Response_fp.h"
#include "CommandDispatcher_fp.h"
#ifdef CC_AC_Send
#   include "AC_spt_fp.h"
#endif // CC_AC_Send
```

Miscellaneous

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

Internal cryptographic functions

```
#include "BnConvert_fp.h"
#include "BnMath_fp.h"
#include "BnMemory_fp.h"
```

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

Support library

```
72  #include "SupportLibraryFunctionPrototypes_fp.h"
```

Linkage to platform functions

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

5.12 LibSupport.h

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

```
#ifndef _LIB_SUPPORT_H_
#define _LIB_SUPPORT_H_
#ifndef RADIX_BITS
#   if defined(__x86_64__) || defined(__x86_64)
\
        || defined(__amd64__) || defined(__amd64) || defined(_WIN64) ||
defined(_M_X64) \
        || defined(_M_ARM64) || defined(__aarch64__)
#       define RADIX_BITS                      64
#   elif defined(__i386__) || defined(__i386) || defined(i386)
\
        || defined(_WIN32) || defined(_M_IX86)
\
        || defined(_M_ARM) || defined(__arm__) || defined(__thumb__)
#       define RADIX_BITS                      32
#   else
#       error Unable to determine RADIX_BITS from compiler environment
#   endif
#endif // RADIX_BITS
```

These macros use the selected libraries to the proper include files.

```
#define LIB_QUOTE(_STRING_) #_STRING_
#define LIB_INCLUDE2(_LIB_, _TYPE_) LIB_QUOTE(_LIB_/TpmTo##_LIB_##_TYPE_.h)
#define LIB_INCLUDE(_LIB_, _TYPE_) LIB_INCLUDE2(_LIB_, _TYPE_)
```

Include the options for hashing and symmetric. Defer the load of the math package Until the bignum parameters are defined.

```
#include LIB_INCLUDE(SYM_LIB, Sym)
#include LIB_INCLUDE(HASH_LIB, Hash)
#undef MIN
#undef MAX
#endif // _LIB_SUPPORT_H_
```

5.13 MinMax.h

```
#ifndef _MIN_MAX_H_
#define _MIN_MAX_H_
#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif
#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif
#endif // _MIN_MAX_H_
```

5.14 NV.h

5.14.1 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

```
#ifndef    _NV_H_
#define    _NV_H_
#ifdef     TPM_NT_ORDINARY
```

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

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

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

```
#   define GetNv_TPM_NV(attributes)                              \
        (   IS_ATTRIBUTE(attributes, TPMA_NV, COUNTER)           \
        +   (IS_ATTRIBUTE(attributes, TPMA_NV, BITS) << 1)       \
        +   (IS_ATTRIBUTE(attributes, TPMA_NV, EXTEND) << 2)     \
        )
#   define TPM_NT_ORDINARY (0)
#   define TPM_NT_COUNTER  (1)
#   define TPM_NT_BITS     (2)
#   define TPM_NT_EXTEND   (4)
#endif
```

5.14.2 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

```
#   define IsNvOrdinaryIndex(attributes)                         \
                (GET_TPM_NT(attributes) == TPM_NT_ORDINARY)
#   define  IsNvCounterIndex(attributes)                         \
                (GET_TPM_NT(attributes) == TPM_NT_COUNTER)
#   define  IsNvBitsIndex(attributes)                            \
               (GET_TPM_NT(attributes) == TPM_NT_BITS)
#   define  IsNvExtendIndex(attributes)                          \
                (GET_TPM_NT(attributes) == TPM_NT_EXTEND)
#ifdef TPM_NT_PIN_PASS
#   define  IsNvPinPassIndex(attributes)                         \
                (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS)
#endif
#ifdef TPM_NT_PIN_FAIL
#   define  IsNvPinFailIndex(attributes)                         \
                (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL)
#endif
typedef struct {
    UINT32      size;
    TPM_HANDLE  handle;
} NV_ENTRY_HEADER;
#define NV_EVICT_OBJECT_SIZE        \
     (sizeof(UINT32)  + sizeof(TPM_HANDLE) + sizeof(OBJECT))
#define NV_INDEX_COUNTER_SIZE       \
    (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
#define NV_RAM_INDEX_COUNTER_SIZE   \
    (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
```

```
42  typedef struct {
43      UINT32          size;
44      TPM_HANDLE      handle;
45      TPMA_NV         attributes;
46  } 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 UINT64 *maxCounter* value.

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

5.14.3 Orderly RAM Values

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

```
48  #define     NV_RAM_REF_INIT         0
```

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

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

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

```
51  #define     NV_ORDERLY_START                \
52                  (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

```
53  #define     RAM_ORDERLY_END                 \
54                  (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.

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

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

```
57  #define ORDERLY_RAM_ADDRESS_OK(start, offset)       \
58          ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))
59  #define RETURN_IF_NV_IS_NOT_AVAILABLE               \
60  {                                                   \
61      if(g_NvStatus != TPM_RC_SUCCESS)                \
62          return g_NvStatus;                          \
63  }
```

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

```
64  #define RETURN_IF_ORDERLY                           \
65  {                                                   \
66      if(NvClearOrderly() != TPM_RC_SUCCESS)          \
67          return g_NvStatus;                          \
68  }
69  #define NV_IS_AVAILABLE     (g_NvStatus == TPM_RC_SUCCESS)
70  #define IS_ORDERLY(value)   (value < SU_DA_USED_VALUE)
71  #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_
```

5.15 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. 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
```

5.16 Tpm.h

Root header file for building any TPM.lib code

```
#ifndef     _TPM_H_
#define     _TPM_H_
#include "TpmBuildSwitches.h"
#include "BaseTypes.h"
#include "TPMB.h"
#include "MinMax.h"
#include "TpmProfile.h"
#include "TpmAlgorithmDefines.h"
#include "LibSupport.h"         // Types from the library. These need to come before
                                // Global.h because some of the structures in
                                // that file depend on the structures used by the
                                // cryptographic libraries.
#include "GpMacros.h"           // Define additional macros
#include "Global.h"             // Define other TPM types
#include "InternalRoutines.h"   // Function prototypes
#endif // _TPM_H_
```

5.17 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 "#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.

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

Allow the command line to specify a **profile** file

```
7   #ifdef PROFILE
8   #   define PROFILE_QUOTE(a) #a
9   #   define PROFILE_INCLUDE(a) PROFILE_QUOTE(a)
10  #   include PROFILE_INCLUDE(PROFILE)
11  #endif
```

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

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

This definition is required for the re-factored code

```
23  #if (!defined USE_BN_ECC_DATA)                                              \
24                  || ((USE_BN_ECC_DATA != NO) && (USE_BN_ECC_DATA != YES))
25  #   undef   USE_BN_ECC_DATA
26  #   define  USE_BN_ECC_DATA             YES     // Default: Either YES or NO
27  #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

```
28  #if !(defined SIMULATION) || ((SIMULATION != NO) && (SIMULATION != YES))
29  #   undef   SIMULATION
30  #   define  SIMULATION                  YES     // Default: Either YES or NO
31  #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.

```
32  #if !(defined LIBRARY_COMPATIBILITY_CHECK)                                      \
33          || (( LIBRARY_COMPATIBILITY_CHECK != NO)                                \
34              && (LIBRARY_COMPATIBILITY_CHECK != YES))
35  #   undef   LIBRARY_COMPATIBILITY_CHECK
36  #   define  LIBRARY_COMPATIBILITY_CHECK YES     // Default: Either YES or NO
37  #endif
38  #if !(defined FIPS_COMPLIANT) || ((FIPS_COMPLIANT != NO) && (FIPS_COMPLIANT != YES))
39  #   undef   FIPS_COMPLIANT
40  #   define  FIPS_COMPLIANT              YES     // Default: Either YES or NO
41  #endif
```

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

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

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

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

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

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

Enable the generation of RSA primes using a sieve.

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

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

```
59  #if RSA_KEY_SIEVE && SIMULATION
60  #   if !(defined RSA_INSTRUMENT)                                                \
61              || ((RSA_INSTRUMENT != NO) && (RSA_INSTRUMENT != YES))
62  #       undef   RSA_INSTRUMENT
63  #       define  RSA_INSTRUMENT          NO      // Default: Either YES or NO
64  #   endif
65  #endif
```

This switch enables the RNG state save and restore

```
#if !(defined _DRBG_STATE_SAVE)                                                 \
    || ((_DRBG_STATE_SAVE != NO) && (_DRBG_STATE_SAVE != YES))
#   undef   _DRBG_STATE_SAVE
#   define  _DRBG_STATE_SAVE            YES     // Default: Either YES or NO
#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.

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

```
#if !(defined CLOCK_STOPS) || ((CLOCK_STOPS != NO) && (CLOCK_STOPS != YES))
#   undef   CLOCK_STOPS
#   define  CLOCK_STOPS                 NO      // Default: Either YES or NO
#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.

```
#if (defined USE_MARSHALING_DEFINES) && (USE_MARSHALING_DEFINES != NO)
#   undef   USE_MARSHALING_DEFINES
#   define  USE_MARSHALING_DEFINES      YES
#else
#   define USE_MARSHALING_DEFINES       YES     // Default: Either YES or NO
#endif
```

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

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

This forces the use of a smaller context slot size. This reduction reduces the range of the epoch allowing the tester to force the epoch to occur faster than the normal defined in TpmProfile.h

```
#   if !(defined CONTEXT_SLOT)
#       define CONTEXT_SLOT            UINT8
#   endif
```

Enables use of the key cache. Default is YES

```
#   if !(defined USE_RSA_KEY_CACHE)                                             \
    || ((USE_RSA_KEY_CACHE != NO) && (USE_RSA_KEY_CACHE != YES))
#       undef   USE_RSA_KEY_CACHE
#       define  USE_RSA_KEY_CACHE       YES     // Default: Either YES or NO
#   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

```
#   if USE_RSA_KEY_CACHE
#       if !(defined USE_KEY_CACHE_FILE)                                        \
            || ((USE_KEY_CACHE_FILE != NO) && (USE_KEY_CACHE_FILE != YES))
#           undef   USE_KEY_CACHE_FILE
#           define  USE_KEY_CACHE_FILE  YES     // Default: Either YES or NO
#       endif
#   else
#       undef   USE_KEY_CACHE_FILE
#       define  USE_KEY_CACHE_FILE      NO
#   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.

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

```
#else
#   define USE_RSA_KEY_CACHE           NO
#   define USE_RSA_KEY_CACHE_FILE      NO
#   define USE_DEBUG_RNG               NO
#endif  // DEBUG && SIMULATION
#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.

```
#   if !(defined COMPILER_CHECKS)                                           \
        || ((COMPILER_CHECKS != NO) && (COMPILER_CHECKS != YES))
#       undef   COMPILER_CHECKS
#       define  COMPILER_CHECKS         NO      // Default: Either YES or NO
#   endif
```

Some of the values (such as sizes) are the result of different options set in TpmProfile.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.

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

```
125 #   if !(defined DRBG_DEBUG_PRINT)                                          \
126     || ((DRBG_DEBUG_PRINT != NO) && (DRBG_DEBUG_PRINT != YES))
127 #       undef   DRBG_DEBUG_PRINT
128 #       define  DRBG_DEBUG_PRINT        NO      // Default: Either YES or NO
129 #   endif
```

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

```
130 #   if !(defined FAIL_TRACE) || ((FAIL_TRACE != NO) && (FAIL_TRACE != YES))
131 #       undef   FAIL_TRACE
132 #       define  FAIL_TRACE              YES     // Default: Either YES or NO
133 #   endif
134 #endif // DEBUG
```

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

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

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

```
140 #if !(defined USE_SPEC_COMPLIANT_PROOFS)                                    \
141     || ((USE_SPEC_COMPLIANT_PROOFS != NO) && (USE_SPEC_COMPLIANT_PROOFS != YES))
142 #   undef   USE_SPEC_COMPLIANT_PROOFS
143 #   define  USE_SPEC_COMPLIANT_PROOFS   YES     // Default: Either YES or NO
144 #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.

```
145 #if !(defined SKIP_PROOF_ERRORS)                                           \
146     || ((SKIP_PROOF_ERRORS != NO) && (SKIP_PROOF_ERRORS != YES))
147 #   undef   SKIP_PROOF_ERRORS
148 #   define  SKIP_PROOF_ERRORS           NO      // Default: Either YES or NO
149 #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*

```
150 #if !(defined USE_BIT_FIELD_STRUCTURES)                                    \
151     || ((USE_BIT_FIELD_STRUCTURES != NO) && (USE_BIT_FIELD_STRUCTURES != YES))
152 #   undef   USE_BIT_FIELD_STRUCTURES
153 #   define  USE_BIT_FIELD_STRUCTURES    DEBUG       // Default: Either YES or NO
154 #endif
```

This define is used to control the debug for the CertifyX509() command.

```
155 #if !(defined CERTIFYX509_DEBUG)                                           \
156     || ((CERTIFYX509_DEBUG != NO) && (CERTIFYX509_DEBUG != YES))
157 #   undef   CERTIFYX509_DEBUG
158 #   define  CERTIFYX509_DEBUG YES               // Default: Either YES or NO
159 #endif
```

This define is used to enable the new table-driven marshaling code.

```
160  #if !(defined TABLE_DRIVEN_MARSHAL)                                    \
161      || ((TABLE_DRIVEN_MARSHAL != NO) && (TABLE_DRIVEN_MARSHAL != YES))
162  #   undef   TABLE_DRIVEN_MARSHAL
163  #   define  TABLE_DRIVEN_MARSHAL YES    // Default: Either YES or NO
164  #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.

```
165  #define      ALG_YES      YES
166  #define      ALG_NO       NO
167  #define      CC_YES       YES
168  #define      CC_NO        NO
169  #endif // _TPM_BUILD_SWITCHES_H_
```

5.18 TpmError.h

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

5.19 TpmTypes.h

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

Table 1:2 - Definition of TPM_ALG_ID Constants

```
typedef UINT16                      TPM_ALG_ID;
#define TYPE_OF_TPM_ALG_ID          UINT16
#define     ALG_ERROR_VALUE         0x0000
#define TPM_ALG_ERROR               (TPM_ALG_ID)(ALG_ERROR_VALUE)
#define     ALG_RSA_VALUE           0x0001
#define TPM_ALG_RSA                 (TPM_ALG_ID)(ALG_RSA_VALUE)
#define     ALG_TDES_VALUE          0x0003
#define TPM_ALG_TDES                (TPM_ALG_ID)(ALG_TDES_VALUE)
#define     ALG_SHA_VALUE           0x0004
#define TPM_ALG_SHA                 (TPM_ALG_ID)(ALG_SHA_VALUE)
#define     ALG_SHA1_VALUE          0x0004
#define TPM_ALG_SHA1                (TPM_ALG_ID)(ALG_SHA1_VALUE)
#define     ALG_HMAC_VALUE          0x0005
#define TPM_ALG_HMAC                (TPM_ALG_ID)(ALG_HMAC_VALUE)
#define     ALG_AES_VALUE           0x0006
#define TPM_ALG_AES                 (TPM_ALG_ID)(ALG_AES_VALUE)
#define     ALG_MGF1_VALUE          0x0007
#define TPM_ALG_MGF1                (TPM_ALG_ID)(ALG_MGF1_VALUE)
#define     ALG_KEYEDHASH_VALUE     0x0008
#define TPM_ALG_KEYEDHASH           (TPM_ALG_ID)(ALG_KEYEDHASH_VALUE)
#define     ALG_XOR_VALUE           0x000A
#define TPM_ALG_XOR                 (TPM_ALG_ID)(ALG_XOR_VALUE)
#define     ALG_SHA256_VALUE        0x000B
#define TPM_ALG_SHA256              (TPM_ALG_ID)(ALG_SHA256_VALUE)
#define     ALG_SHA384_VALUE        0x000C
#define TPM_ALG_SHA384              (TPM_ALG_ID)(ALG_SHA384_VALUE)
#define     ALG_SHA512_VALUE        0x000D
#define TPM_ALG_SHA512              (TPM_ALG_ID)(ALG_SHA512_VALUE)
#define     ALG_NULL_VALUE          0x0010
#define TPM_ALG_NULL                (TPM_ALG_ID)(ALG_NULL_VALUE)
#define     ALG_SM3_256_VALUE       0x0012
#define TPM_ALG_SM3_256             (TPM_ALG_ID)(ALG_SM3_256_VALUE)
#define     ALG_SM4_VALUE           0x0013
#define TPM_ALG_SM4                 (TPM_ALG_ID)(ALG_SM4_VALUE)
#define     ALG_RSASSA_VALUE        0x0014
#define TPM_ALG_RSASSA              (TPM_ALG_ID)(ALG_RSASSA_VALUE)
#define     ALG_RSAES_VALUE         0x0015
#define TPM_ALG_RSAES               (TPM_ALG_ID)(ALG_RSAES_VALUE)
#define     ALG_RSAPSS_VALUE        0x0016
#define TPM_ALG_RSAPSS              (TPM_ALG_ID)(ALG_RSAPSS_VALUE)
#define     ALG_OAEP_VALUE          0x0017
#define TPM_ALG_OAEP                (TPM_ALG_ID)(ALG_OAEP_VALUE)
#define     ALG_ECDSA_VALUE         0x0018
#define TPM_ALG_ECDSA               (TPM_ALG_ID)(ALG_ECDSA_VALUE)
#define     ALG_ECDH_VALUE          0x0019
#define TPM_ALG_ECDH                (TPM_ALG_ID)(ALG_ECDH_VALUE)
#define     ALG_ECDAA_VALUE         0x001A
#define TPM_ALG_ECDAA               (TPM_ALG_ID)(ALG_ECDAA_VALUE)
#define     ALG_SM2_VALUE           0x001B
#define TPM_ALG_SM2                 (TPM_ALG_ID)(ALG_SM2_VALUE)
#define     ALG_ECSCHNORR_VALUE     0x001C
#define TPM_ALG_ECSCHNORR           (TPM_ALG_ID)(ALG_ECSCHNORR_VALUE)
#define     ALG_ECMQV_VALUE         0x001D
#define TPM_ALG_ECMQV               (TPM_ALG_ID)(ALG_ECMQV_VALUE)
#define     ALG_KDF1_SP800_56A_VALUE 0x0020
#define TPM_ALG_KDF1_SP800_56A      (TPM_ALG_ID)(ALG_KDF1_SP800_56A_VALUE)
#define     ALG_KDF2_VALUE          0x0021
```

```
#define TPM_ALG_KDF2                    (TPM_ALG_ID)(ALG_KDF2_VALUE)
#define     ALG_KDF1_SP800_108_VALUE    0x0022
#define TPM_ALG_KDF1_SP800_108          (TPM_ALG_ID)(ALG_KDF1_SP800_108_VALUE)
#define     ALG_ECC_VALUE               0x0023
#define TPM_ALG_ECC                     (TPM_ALG_ID)(ALG_ECC_VALUE)
#define     ALG_SYMCIPHER_VALUE         0x0025
#define TPM_ALG_SYMCIPHER               (TPM_ALG_ID)(ALG_SYMCIPHER_VALUE)
#define     ALG_CAMELLIA_VALUE          0x0026
#define TPM_ALG_CAMELLIA                (TPM_ALG_ID)(ALG_CAMELLIA_VALUE)
#define     ALG_SHA3_256_VALUE          0x0027
#define TPM_ALG_SHA3_256                (TPM_ALG_ID)(ALG_SHA3_256_VALUE)
#define     ALG_SHA3_384_VALUE          0x0028
#define TPM_ALG_SHA3_384                (TPM_ALG_ID)(ALG_SHA3_384_VALUE)
#define     ALG_SHA3_512_VALUE          0x0029
#define TPM_ALG_SHA3_512                (TPM_ALG_ID)(ALG_SHA3_512_VALUE)
#define     ALG_CMAC_VALUE              0x003F
#define TPM_ALG_CMAC                    (TPM_ALG_ID)(ALG_CMAC_VALUE)
#define     ALG_CTR_VALUE               0x0040
#define TPM_ALG_CTR                     (TPM_ALG_ID)(ALG_CTR_VALUE)
#define     ALG_OFB_VALUE               0x0041
#define TPM_ALG_OFB                     (TPM_ALG_ID)(ALG_OFB_VALUE)
#define     ALG_CBC_VALUE               0x0042
#define TPM_ALG_CBC                     (TPM_ALG_ID)(ALG_CBC_VALUE)
#define     ALG_CFB_VALUE               0x0043
#define TPM_ALG_CFB                     (TPM_ALG_ID)(ALG_CFB_VALUE)
#define     ALG_ECB_VALUE               0x0044
#define TPM_ALG_ECB                     (TPM_ALG_ID)(ALG_ECB_VALUE)
```

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)
```

Table 1:4 - Definition of TPM_ECC_CURVE Constants

```
typedef UINT16              TPM_ECC_CURVE;
#define TYPE_OF_TPM_ECC_CURVE   UINT16
#define TPM_ECC_NONE        (TPM_ECC_CURVE)(0x0000)
#define TPM_ECC_NIST_P192   (TPM_ECC_CURVE)(0x0001)
#define TPM_ECC_NIST_P224   (TPM_ECC_CURVE)(0x0002)
#define TPM_ECC_NIST_P256   (TPM_ECC_CURVE)(0x0003)
#define TPM_ECC_NIST_P384   (TPM_ECC_CURVE)(0x0004)
#define TPM_ECC_NIST_P521   (TPM_ECC_CURVE)(0x0005)
#define TPM_ECC_BN_P256     (TPM_ECC_CURVE)(0x0010)
#define TPM_ECC_BN_P638     (TPM_ECC_CURVE)(0x0011)
#define TPM_ECC_SM2_P256    (TPM_ECC_CURVE)(0x0020)
```

Table 2:12 - Definition of TPM_CC Constants

```
typedef UINT32                          TPM_CC;
#define TYPE_OF_TPM_CC                  UINT32
#define TPM_CC_NV_UndefineSpaceSpecial  (TPM_CC)(0x0000011F)
#define TPM_CC_EvictControl             (TPM_CC)(0x00000120)
#define TPM_CC_HierarchyControl         (TPM_CC)(0x00000121)
#define TPM_CC_NV_UndefineSpace         (TPM_CC)(0x00000122)
#define TPM_CC_ChangeEPS                (TPM_CC)(0x00000124)
#define TPM_CC_ChangePPS                (TPM_CC)(0x00000125)
#define TPM_CC_Clear                    (TPM_CC)(0x00000126)
#define TPM_CC_ClearControl             (TPM_CC)(0x00000127)
#define TPM_CC_ClockSet                 (TPM_CC)(0x00000128)
#define TPM_CC_HierarchyChangeAuth      (TPM_CC)(0x00000129)
#define TPM_CC_NV_DefineSpace           (TPM_CC)(0x0000012A)
```

```
115  #define TPM_CC_PCR_Allocate                (TPM_CC)(0x0000012B)
116  #define TPM_CC_PCR_SetAuthPolicy           (TPM_CC)(0x0000012C)
117  #define TPM_CC_PP_Commands                 (TPM_CC)(0x0000012D)
118  #define TPM_CC_SetPrimaryPolicy            (TPM_CC)(0x0000012E)
119  #define TPM_CC_FieldUpgradeStart           (TPM_CC)(0x0000012F)
120  #define TPM_CC_ClockRateAdjust             (TPM_CC)(0x00000130)
121  #define TPM_CC_CreatePrimary               (TPM_CC)(0x00000131)
122  #define TPM_CC_NV_GlobalWriteLock          (TPM_CC)(0x00000132)
123  #define TPM_CC_GetCommandAuditDigest       (TPM_CC)(0x00000133)
124  #define TPM_CC_NV_Increment                (TPM_CC)(0x00000134)
125  #define TPM_CC_NV_SetBits                  (TPM_CC)(0x00000135)
126  #define TPM_CC_NV_Extend                   (TPM_CC)(0x00000136)
127  #define TPM_CC_NV_Write                    (TPM_CC)(0x00000137)
128  #define TPM_CC_NV_WriteLock                (TPM_CC)(0x00000138)
129  #define TPM_CC_DictionaryAttackLockReset   (TPM_CC)(0x00000139)
130  #define TPM_CC_DictionaryAttackParameters  (TPM_CC)(0x0000013A)
131  #define TPM_CC_NV_ChangeAuth               (TPM_CC)(0x0000013B)
132  #define TPM_CC_PCR_Event                   (TPM_CC)(0x0000013C)
133  #define TPM_CC_PCR_Reset                   (TPM_CC)(0x0000013D)
134  #define TPM_CC_SequenceComplete            (TPM_CC)(0x0000013E)
135  #define TPM_CC_SetAlgorithmSet             (TPM_CC)(0x0000013F)
136  #define TPM_CC_SetCommandCodeAuditStatus   (TPM_CC)(0x00000140)
137  #define TPM_CC_FieldUpgradeData            (TPM_CC)(0x00000141)
138  #define TPM_CC_IncrementalSelfTest         (TPM_CC)(0x00000142)
139  #define TPM_CC_SelfTest                    (TPM_CC)(0x00000143)
140  #define TPM_CC_Startup                     (TPM_CC)(0x00000144)
141  #define TPM_CC_Shutdown                    (TPM_CC)(0x00000145)
142  #define TPM_CC_StirRandom                  (TPM_CC)(0x00000146)
143  #define TPM_CC_ActivateCredential          (TPM_CC)(0x00000147)
144  #define TPM_CC_Certify                     (TPM_CC)(0x00000148)
145  #define TPM_CC_PolicyNV                    (TPM_CC)(0x00000149)
146  #define TPM_CC_CertifyCreation             (TPM_CC)(0x0000014A)
147  #define TPM_CC_Duplicate                   (TPM_CC)(0x0000014B)
148  #define TPM_CC_GetTime                     (TPM_CC)(0x0000014C)
149  #define TPM_CC_GetSessionAuditDigest       (TPM_CC)(0x0000014D)
150  #define TPM_CC_NV_Read                     (TPM_CC)(0x0000014E)
151  #define TPM_CC_NV_ReadLock                 (TPM_CC)(0x0000014F)
152  #define TPM_CC_ObjectChangeAuth            (TPM_CC)(0x00000150)
153  #define TPM_CC_PolicySecret                (TPM_CC)(0x00000151)
154  #define TPM_CC_Rewrap                      (TPM_CC)(0x00000152)
155  #define TPM_CC_Create                      (TPM_CC)(0x00000153)
156  #define TPM_CC_ECDH_ZGen                   (TPM_CC)(0x00000154)
157  #define TPM_CC_HMAC                        (TPM_CC)(0x00000155)
158  #define TPM_CC_MAC                         (TPM_CC)(0x00000155)
159  #define TPM_CC_Import                      (TPM_CC)(0x00000156)
160  #define TPM_CC_Load                        (TPM_CC)(0x00000157)
161  #define TPM_CC_Quote                       (TPM_CC)(0x00000158)
162  #define TPM_CC_RSA_Decrypt                 (TPM_CC)(0x00000159)
163  #define TPM_CC_HMAC_Start                  (TPM_CC)(0x0000015B)
164  #define TPM_CC_MAC_Start                   (TPM_CC)(0x0000015B)
165  #define TPM_CC_SequenceUpdate              (TPM_CC)(0x0000015C)
166  #define TPM_CC_Sign                        (TPM_CC)(0x0000015D)
167  #define TPM_CC_Unseal                      (TPM_CC)(0x0000015E)
168  #define TPM_CC_PolicySigned                (TPM_CC)(0x00000160)
169  #define TPM_CC_ContextLoad                 (TPM_CC)(0x00000161)
170  #define TPM_CC_ContextSave                 (TPM_CC)(0x00000162)
171  #define TPM_CC_ECDH_KeyGen                 (TPM_CC)(0x00000163)
172  #define TPM_CC_EncryptDecrypt              (TPM_CC)(0x00000164)
173  #define TPM_CC_FlushContext                (TPM_CC)(0x00000165)
174  #define TPM_CC_LoadExternal                (TPM_CC)(0x00000167)
175  #define TPM_CC_MakeCredential              (TPM_CC)(0x00000168)
176  #define TPM_CC_NV_ReadPublic               (TPM_CC)(0x00000169)
177  #define TPM_CC_PolicyAuthorize             (TPM_CC)(0x0000016A)
178  #define TPM_CC_PolicyAuthValue             (TPM_CC)(0x0000016B)
179  #define TPM_CC_PolicyCommandCode           (TPM_CC)(0x0000016C)
180  #define TPM_CC_PolicyCounterTimer          (TPM_CC)(0x0000016D)
```

```
181   #define TPM_CC_PolicyCpHash                (TPM_CC)(0x0000016E)
182   #define TPM_CC_PolicyLocality              (TPM_CC)(0x0000016F)
183   #define TPM_CC_PolicyNameHash              (TPM_CC)(0x00000170)
184   #define TPM_CC_PolicyOR                    (TPM_CC)(0x00000171)
185   #define TPM_CC_PolicyTicket                (TPM_CC)(0x00000172)
186   #define TPM_CC_ReadPublic                  (TPM_CC)(0x00000173)
187   #define TPM_CC_RSA_Encrypt                 (TPM_CC)(0x00000174)
188   #define TPM_CC_StartAuthSession            (TPM_CC)(0x00000176)
189   #define TPM_CC_VerifySignature             (TPM_CC)(0x00000177)
190   #define TPM_CC_ECC_Parameters              (TPM_CC)(0x00000178)
191   #define TPM_CC_FirmwareRead                (TPM_CC)(0x00000179)
192   #define TPM_CC_GetCapability               (TPM_CC)(0x0000017A)
193   #define TPM_CC_GetRandom                   (TPM_CC)(0x0000017B)
194   #define TPM_CC_GetTestResult               (TPM_CC)(0x0000017C)
195   #define TPM_CC_Hash                        (TPM_CC)(0x0000017D)
196   #define TPM_CC_PCR_Read                    (TPM_CC)(0x0000017E)
197   #define TPM_CC_PolicyPCR                   (TPM_CC)(0x0000017F)
198   #define TPM_CC_PolicyRestart               (TPM_CC)(0x00000180)
199   #define TPM_CC_ReadClock                   (TPM_CC)(0x00000181)
200   #define TPM_CC_PCR_Extend                  (TPM_CC)(0x00000182)
201   #define TPM_CC_PCR_SetAuthValue            (TPM_CC)(0x00000183)
202   #define TPM_CC_NV_Certify                  (TPM_CC)(0x00000184)
203   #define TPM_CC_EventSequenceComplete       (TPM_CC)(0x00000185)
204   #define TPM_CC_HashSequenceStart           (TPM_CC)(0x00000186)
205   #define TPM_CC_PolicyPhysicalPresence      (TPM_CC)(0x00000187)
206   #define TPM_CC_PolicyDuplicationSelect     (TPM_CC)(0x00000188)
207   #define TPM_CC_PolicyGetDigest             (TPM_CC)(0x00000189)
208   #define TPM_CC_TestParms                   (TPM_CC)(0x0000018A)
209   #define TPM_CC_Commit                      (TPM_CC)(0x0000018B)
210   #define TPM_CC_PolicyPassword              (TPM_CC)(0x0000018C)
211   #define TPM_CC_ZGen_2Phase                 (TPM_CC)(0x0000018D)
212   #define TPM_CC_EC_Ephemeral                (TPM_CC)(0x0000018E)
213   #define TPM_CC_PolicyNvWritten             (TPM_CC)(0x0000018F)
214   #define TPM_CC_PolicyTemplate              (TPM_CC)(0x00000190)
215   #define TPM_CC_CreateLoaded                (TPM_CC)(0x00000191)
216   #define TPM_CC_PolicyAuthorizeNV           (TPM_CC)(0x00000192)
217   #define TPM_CC_EncryptDecrypt2             (TPM_CC)(0x00000193)
218   #define TPM_CC_AC_GetCapability            (TPM_CC)(0x00000194)
219   #define TPM_CC_AC_Send                     (TPM_CC)(0x00000195)
220   #define TPM_CC_Policy_AC_SendSelect        (TPM_CC)(0x00000196)
221   #define TPM_CC_CertifyX509                 (TPM_CC)(0x00000197)
222   #define TPM_CC_ACT_SetTimeout              (TPM_CC)(0x00000198)
223   #define CC_VEND                            0x20000000
224   #define TPM_CC_Vendor_TCG_Test             (TPM_CC)(0x20000000)
```

Table 2:5 - Definition of Types for Documentation Clarity

```
225   typedef UINT32              TPM_ALGORITHM_ID;
226   #define TYPE_OF_TPM_ALGORITHM_ID    UINT32
227   typedef UINT32              TPM_MODIFIER_INDICATOR;
228   #define TYPE_OF_TPM_MODIFIER_INDICATOR  UINT32
229   typedef UINT32              TPM_AUTHORIZATION_SIZE;
230   #define TYPE_OF_TPM_AUTHORIZATION_SIZE  UINT32
231   typedef UINT32              TPM_PARAMETER_SIZE;
232   #define TYPE_OF_TPM_PARAMETER_SIZE  UINT32
233   typedef UINT16              TPM_KEY_SIZE;
234   #define TYPE_OF_TPM_KEY_SIZE    UINT16
235   typedef UINT16              TPM_KEY_BITS;
236   #define TYPE_OF_TPM_KEY_BITS    UINT16
```

Table 2:6 - Definition of TPM_SPEC Constants

```
237   typedef UINT32                TPM_SPEC;
238   #define TYPE_OF_TPM_SPEC      UINT32
239   #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             159
#define TPM_SPEC_VERSION         (TPM_SPEC)(SPEC_VERSION)
#define SPEC_YEAR                2019
#define TPM_SPEC_YEAR            (TPM_SPEC)(SPEC_YEAR)
#define SPEC_DAY_OF_YEAR         312
#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 TYPE_OF_TPM_GENERATED    UINT32
#define TPM_GENERATED_VALUE      (TPM_GENERATED)(0xFF544347)
```

Table 2:16 - Definition of TPM_RC Constants

```
typedef UINT32                       TPM_RC;
#define TYPE_OF_TPM_RC               UINT32
#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)
#define TPM_RC_NV_AUTHORIZATION      (TPM_RC)(RC_VER1+0x049)
#define TPM_RC_NV_UNINITIALIZED      (TPM_RC)(RC_VER1+0x04A)
#define TPM_RC_NV_SPACE              (TPM_RC)(RC_VER1+0x04B)
#define TPM_RC_NV_DEFINED            (TPM_RC)(RC_VER1+0x04C)
#define TPM_RC_BAD_CONTEXT           (TPM_RC)(RC_VER1+0x050)
#define TPM_RC_CPHASH                (TPM_RC)(RC_VER1+0x051)
#define TPM_RC_PARENT                (TPM_RC)(RC_VER1+0x052)
#define TPM_RC_NEEDS_TEST            (TPM_RC)(RC_VER1+0x053)
#define TPM_RC_NO_RESULT             (TPM_RC)(RC_VER1+0x054)
#define TPM_RC_SENSITIVE             (TPM_RC)(RC_VER1+0x055)
#define RC_MAX_FM0                   (TPM_RC)(RC_VER1+0x07F)
#define RC_FMT1                      (TPM_RC)(0x080)
#define TPM_RC_ASYMMETRIC            (TPM_RC)(RC_FMT1+0x001)
#define TPM_RCS_ASYMMETRIC           (TPM_RC)(RC_FMT1+0x001)
#define TPM_RC_ATTRIBUTES            (TPM_RC)(RC_FMT1+0x002)
#define TPM_RCS_ATTRIBUTES           (TPM_RC)(RC_FMT1+0x002)
#define TPM_RC_HASH                  (TPM_RC)(RC_FMT1+0x003)
#define TPM_RCS_HASH                 (TPM_RC)(RC_FMT1+0x003)
```



```
299   #define TPM_RC_VALUE              (TPM_RC)(RC_FMT1+0x004)
300   #define TPM_RCS_VALUE             (TPM_RC)(RC_FMT1+0x004)
301   #define TPM_RC_HIERARCHY          (TPM_RC)(RC_FMT1+0x005)
302   #define TPM_RCS_HIERARCHY         (TPM_RC)(RC_FMT1+0x005)
303   #define TPM_RC_KEY_SIZE           (TPM_RC)(RC_FMT1+0x007)
304   #define TPM_RCS_KEY_SIZE          (TPM_RC)(RC_FMT1+0x007)
305   #define TPM_RC_MGF                (TPM_RC)(RC_FMT1+0x008)
306   #define TPM_RCS_MGF               (TPM_RC)(RC_FMT1+0x008)
307   #define TPM_RC_MODE               (TPM_RC)(RC_FMT1+0x009)
308   #define TPM_RCS_MODE              (TPM_RC)(RC_FMT1+0x009)
309   #define TPM_RC_TYPE               (TPM_RC)(RC_FMT1+0x00A)
310   #define TPM_RCS_TYPE              (TPM_RC)(RC_FMT1+0x00A)
311   #define TPM_RC_HANDLE             (TPM_RC)(RC_FMT1+0x00B)
312   #define TPM_RCS_HANDLE            (TPM_RC)(RC_FMT1+0x00B)
313   #define TPM_RC_KDF                (TPM_RC)(RC_FMT1+0x00C)
314   #define TPM_RCS_KDF               (TPM_RC)(RC_FMT1+0x00C)
315   #define TPM_RC_RANGE              (TPM_RC)(RC_FMT1+0x00D)
316   #define TPM_RCS_RANGE             (TPM_RC)(RC_FMT1+0x00D)
317   #define TPM_RC_AUTH_FAIL          (TPM_RC)(RC_FMT1+0x00E)
318   #define TPM_RCS_AUTH_FAIL         (TPM_RC)(RC_FMT1+0x00E)
319   #define TPM_RC_NONCE              (TPM_RC)(RC_FMT1+0x00F)
320   #define TPM_RCS_NONCE             (TPM_RC)(RC_FMT1+0x00F)
321   #define TPM_RC_PP                 (TPM_RC)(RC_FMT1+0x010)
322   #define TPM_RCS_PP                (TPM_RC)(RC_FMT1+0x010)
323   #define TPM_RC_SCHEME             (TPM_RC)(RC_FMT1+0x012)
324   #define TPM_RCS_SCHEME            (TPM_RC)(RC_FMT1+0x012)
325   #define TPM_RC_SIZE               (TPM_RC)(RC_FMT1+0x015)
326   #define TPM_RCS_SIZE              (TPM_RC)(RC_FMT1+0x015)
327   #define TPM_RC_SYMMETRIC          (TPM_RC)(RC_FMT1+0x016)
328   #define TPM_RCS_SYMMETRIC         (TPM_RC)(RC_FMT1+0x016)
329   #define TPM_RC_TAG                (TPM_RC)(RC_FMT1+0x017)
330   #define TPM_RCS_TAG               (TPM_RC)(RC_FMT1+0x017)
331   #define TPM_RC_SELECTOR           (TPM_RC)(RC_FMT1+0x018)
332   #define TPM_RCS_SELECTOR          (TPM_RC)(RC_FMT1+0x018)
333   #define TPM_RC_INSUFFICIENT       (TPM_RC)(RC_FMT1+0x01A)
334   #define TPM_RCS_INSUFFICIENT      (TPM_RC)(RC_FMT1+0x01A)
335   #define TPM_RC_SIGNATURE          (TPM_RC)(RC_FMT1+0x01B)
336   #define TPM_RCS_SIGNATURE         (TPM_RC)(RC_FMT1+0x01B)
337   #define TPM_RC_KEY                (TPM_RC)(RC_FMT1+0x01C)
338   #define TPM_RCS_KEY               (TPM_RC)(RC_FMT1+0x01C)
339   #define TPM_RC_POLICY_FAIL        (TPM_RC)(RC_FMT1+0x01D)
340   #define TPM_RCS_POLICY_FAIL       (TPM_RC)(RC_FMT1+0x01D)
341   #define TPM_RC_INTEGRITY          (TPM_RC)(RC_FMT1+0x01F)
342   #define TPM_RCS_INTEGRITY         (TPM_RC)(RC_FMT1+0x01F)
343   #define TPM_RC_TICKET             (TPM_RC)(RC_FMT1+0x020)
344   #define TPM_RCS_TICKET            (TPM_RC)(RC_FMT1+0x020)
345   #define TPM_RC_RESERVED_BITS      (TPM_RC)(RC_FMT1+0x021)
346   #define TPM_RCS_RESERVED_BITS     (TPM_RC)(RC_FMT1+0x021)
347   #define TPM_RC_BAD_AUTH           (TPM_RC)(RC_FMT1+0x022)
348   #define TPM_RCS_BAD_AUTH          (TPM_RC)(RC_FMT1+0x022)
349   #define TPM_RC_EXPIRED            (TPM_RC)(RC_FMT1+0x023)
350   #define TPM_RCS_EXPIRED           (TPM_RC)(RC_FMT1+0x023)
351   #define TPM_RC_POLICY_CC          (TPM_RC)(RC_FMT1+0x024)
352   #define TPM_RCS_POLICY_CC         (TPM_RC)(RC_FMT1+0x024)
353   #define TPM_RC_BINDING            (TPM_RC)(RC_FMT1+0x025)
354   #define TPM_RCS_BINDING           (TPM_RC)(RC_FMT1+0x025)
355   #define TPM_RC_CURVE              (TPM_RC)(RC_FMT1+0x026)
356   #define TPM_RCS_CURVE             (TPM_RC)(RC_FMT1+0x026)
357   #define TPM_RC_ECC_POINT          (TPM_RC)(RC_FMT1+0x027)
358   #define TPM_RCS_ECC_POINT         (TPM_RC)(RC_FMT1+0x027)
359   #define RC_WARN                   (TPM_RC)(0x900)
360   #define TPM_RC_CONTEXT_GAP        (TPM_RC)(RC_WARN+0x001)
361   #define TPM_RC_OBJECT_MEMORY      (TPM_RC)(RC_WARN+0x002)
362   #define TPM_RC_SESSION_MEMORY     (TPM_RC)(RC_WARN+0x003)
363   #define TPM_RC_MEMORY             (TPM_RC)(RC_WARN+0x004)
364   #define TPM_RC_SESSION_HANDLES    (TPM_RC)(RC_WARN+0x005)
```

```
365  #define TPM_RC_OBJECT_HANDLES       (TPM_RC)(RC_WARN+0x006)
366  #define TPM_RC_LOCALITY             (TPM_RC)(RC_WARN+0x007)
367  #define TPM_RC_YIELDED              (TPM_RC)(RC_WARN+0x008)
368  #define TPM_RC_CANCELED             (TPM_RC)(RC_WARN+0x009)
369  #define TPM_RC_TESTING              (TPM_RC)(RC_WARN+0x00A)
370  #define TPM_RC_REFERENCE_H0         (TPM_RC)(RC_WARN+0x010)
371  #define TPM_RC_REFERENCE_H1         (TPM_RC)(RC_WARN+0x011)
372  #define TPM_RC_REFERENCE_H2         (TPM_RC)(RC_WARN+0x012)
373  #define TPM_RC_REFERENCE_H3         (TPM_RC)(RC_WARN+0x013)
374  #define TPM_RC_REFERENCE_H4         (TPM_RC)(RC_WARN+0x014)
375  #define TPM_RC_REFERENCE_H5         (TPM_RC)(RC_WARN+0x015)
376  #define TPM_RC_REFERENCE_H6         (TPM_RC)(RC_WARN+0x016)
377  #define TPM_RC_REFERENCE_S0         (TPM_RC)(RC_WARN+0x018)
378  #define TPM_RC_REFERENCE_S1         (TPM_RC)(RC_WARN+0x019)
379  #define TPM_RC_REFERENCE_S2         (TPM_RC)(RC_WARN+0x01A)
380  #define TPM_RC_REFERENCE_S3         (TPM_RC)(RC_WARN+0x01B)
381  #define TPM_RC_REFERENCE_S4         (TPM_RC)(RC_WARN+0x01C)
382  #define TPM_RC_REFERENCE_S5         (TPM_RC)(RC_WARN+0x01D)
383  #define TPM_RC_REFERENCE_S6         (TPM_RC)(RC_WARN+0x01E)
384  #define TPM_RC_NV_RATE              (TPM_RC)(RC_WARN+0x020)
385  #define TPM_RC_LOCKOUT              (TPM_RC)(RC_WARN+0x021)
386  #define TPM_RC_RETRY                (TPM_RC)(RC_WARN+0x022)
387  #define TPM_RC_NV_UNAVAILABLE       (TPM_RC)(RC_WARN+0x023)
388  #define TPM_RC_NOT_USED             (TPM_RC)(RC_WARN+0x7F)
389  #define TPM_RC_H                    (TPM_RC)(0x000)
390  #define TPM_RC_P                    (TPM_RC)(0x040)
391  #define TPM_RC_S                    (TPM_RC)(0x800)
392  #define TPM_RC_1                    (TPM_RC)(0x100)
393  #define TPM_RC_2                    (TPM_RC)(0x200)
394  #define TPM_RC_3                    (TPM_RC)(0x300)
395  #define TPM_RC_4                    (TPM_RC)(0x400)
396  #define TPM_RC_5                    (TPM_RC)(0x500)
397  #define TPM_RC_6                    (TPM_RC)(0x600)
398  #define TPM_RC_7                    (TPM_RC)(0x700)
399  #define TPM_RC_8                    (TPM_RC)(0x800)
400  #define TPM_RC_9                    (TPM_RC)(0x900)
401  #define TPM_RC_A                    (TPM_RC)(0xA00)
402  #define TPM_RC_B                    (TPM_RC)(0xB00)
403  #define TPM_RC_C                    (TPM_RC)(0xC00)
404  #define TPM_RC_D                    (TPM_RC)(0xD00)
405  #define TPM_RC_E                    (TPM_RC)(0xE00)
406  #define TPM_RC_F                    (TPM_RC)(0xF00)
407  #define TPM_RC_N_MASK               (TPM_RC)(0xF00)
```

Table 2:17 - Definition of TPM_CLOCK_ADJUST Constants

```
408  typedef INT8                        TPM_CLOCK_ADJUST;
409  #define TYPE_OF_TPM_CLOCK_ADJUST    UINT8
410  #define TPM_CLOCK_COARSE_SLOWER     (TPM_CLOCK_ADJUST)(-3)
411  #define TPM_CLOCK_MEDIUM_SLOWER     (TPM_CLOCK_ADJUST)(-2)
412  #define TPM_CLOCK_FINE_SLOWER       (TPM_CLOCK_ADJUST)(-1)
413  #define TPM_CLOCK_NO_CHANGE         (TPM_CLOCK_ADJUST)(0)
414  #define TPM_CLOCK_FINE_FASTER       (TPM_CLOCK_ADJUST)(1)
415  #define TPM_CLOCK_MEDIUM_FASTER     (TPM_CLOCK_ADJUST)(2)
416  #define TPM_CLOCK_COARSE_FASTER     (TPM_CLOCK_ADJUST)(3)
```

Table 2:18 - Definition of TPM_EO Constants

```
417  typedef UINT16              TPM_EO;
418  #define TYPE_OF_TPM_EO      UINT16
419  #define TPM_EO_EQ           (TPM_EO)(0x0000)
420  #define TPM_EO_NEQ          (TPM_EO)(0x0001)
421  #define TPM_EO_SIGNED_GT    (TPM_EO)(0x0002)
422  #define TPM_EO_UNSIGNED_GT  (TPM_EO)(0x0003)
423  #define TPM_EO_SIGNED_LT    (TPM_EO)(0x0004)
```

```
424  #define TPM_EO_UNSIGNED_LT   (TPM_EO)(0x0005)
425  #define TPM_EO_SIGNED_GE     (TPM_EO)(0x0006)
426  #define TPM_EO_UNSIGNED_GE   (TPM_EO)(0x0007)
427  #define TPM_EO_SIGNED_LE     (TPM_EO)(0x0008)
428  #define TPM_EO_UNSIGNED_LE   (TPM_EO)(0x0009)
429  #define TPM_EO_BITSET        (TPM_EO)(0x000A)
430  #define TPM_EO_BITCLEAR      (TPM_EO)(0x000B)
```

Table 2:19 - Definition of TPM_ST Constants

```
431  typedef UINT16                          TPM_ST;
432  #define TYPE_OF_TPM_ST                  UINT16
433  #define TPM_ST_RSP_COMMAND              (TPM_ST)(0x00C4)
434  #define TPM_ST_NULL                     (TPM_ST)(0x8000)
435  #define TPM_ST_NO_SESSIONS              (TPM_ST)(0x8001)
436  #define TPM_ST_SESSIONS                 (TPM_ST)(0x8002)
437  #define TPM_ST_ATTEST_NV                (TPM_ST)(0x8014)
438  #define TPM_ST_ATTEST_COMMAND_AUDIT     (TPM_ST)(0x8015)
439  #define TPM_ST_ATTEST_SESSION_AUDIT     (TPM_ST)(0x8016)
440  #define TPM_ST_ATTEST_CERTIFY           (TPM_ST)(0x8017)
441  #define TPM_ST_ATTEST_QUOTE             (TPM_ST)(0x8018)
442  #define TPM_ST_ATTEST_TIME              (TPM_ST)(0x8019)
443  #define TPM_ST_ATTEST_CREATION          (TPM_ST)(0x801A)
444  #define TPM_ST_ATTEST_NV_DIGEST         (TPM_ST)(0x801C)
445  #define TPM_ST_CREATION                 (TPM_ST)(0x8021)
446  #define TPM_ST_VERIFIED                 (TPM_ST)(0x8022)
447  #define TPM_ST_AUTH_SECRET              (TPM_ST)(0x8023)
448  #define TPM_ST_HASHCHECK                (TPM_ST)(0x8024)
449  #define TPM_ST_AUTH_SIGNED              (TPM_ST)(0x8025)
450  #define TPM_ST_FU_MANIFEST              (TPM_ST)(0x8029)
```

Table 2:20 - Definition of TPM_SU Constants

```
451  typedef UINT16              TPM_SU;
452  #define TYPE_OF_TPM_SU      UINT16
453  #define TPM_SU_CLEAR        (TPM_SU)(0x0000)
454  #define TPM_SU_STATE        (TPM_SU)(0x0001)
```

Table 2:21 - Definition of TPM_SE Constants

```
455  typedef UINT8               TPM_SE;
456  #define TYPE_OF_TPM_SE      UINT8
457  #define TPM_SE_HMAC         (TPM_SE)(0x00)
458  #define TPM_SE_POLICY       (TPM_SE)(0x01)
459  #define TPM_SE_TRIAL        (TPM_SE)(0x03)
```

Table 2:22 - Definition of TPM_CAP Constants

```
460  typedef UINT32                      TPM_CAP;
461  #define TYPE_OF_TPM_CAP             UINT32
462  #define TPM_CAP_FIRST               (TPM_CAP)(0x00000000)
463  #define TPM_CAP_ALGS                (TPM_CAP)(0x00000000)
464  #define TPM_CAP_HANDLES             (TPM_CAP)(0x00000001)
465  #define TPM_CAP_COMMANDS            (TPM_CAP)(0x00000002)
466  #define TPM_CAP_PP_COMMANDS         (TPM_CAP)(0x00000003)
467  #define TPM_CAP_AUDIT_COMMANDS      (TPM_CAP)(0x00000004)
468  #define TPM_CAP_PCRS                (TPM_CAP)(0x00000005)
469  #define TPM_CAP_TPM_PROPERTIES      (TPM_CAP)(0x00000006)
470  #define TPM_CAP_PCR_PROPERTIES      (TPM_CAP)(0x00000007)
471  #define TPM_CAP_ECC_CURVES          (TPM_CAP)(0x00000008)
472  #define TPM_CAP_AUTH_POLICIES       (TPM_CAP)(0x00000009)
473  #define TPM_CAP_ACT                 (TPM_CAP)(0x0000000A)
474  #define TPM_CAP_LAST                (TPM_CAP)(0x0000000A)
475  #define TPM_CAP_VENDOR_PROPERTY     (TPM_CAP)(0x00000100)
```

Table 2:23 - Definition of TPM_PT Constants

```
476   typedef UINT32                      TPM_PT;
477   #define TYPE_OF_TPM_PT              UINT32
478   #define TPM_PT_NONE                 (TPM_PT)(0x00000000)
479   #define PT_GROUP                    (TPM_PT)(0x00000100)
480   #define PT_FIXED                    (TPM_PT)(PT_GROUP*1)
481   #define TPM_PT_FAMILY_INDICATOR     (TPM_PT)(PT_FIXED+0)
482   #define TPM_PT_LEVEL                (TPM_PT)(PT_FIXED+1)
483   #define TPM_PT_REVISION             (TPM_PT)(PT_FIXED+2)
484   #define TPM_PT_DAY_OF_YEAR          (TPM_PT)(PT_FIXED+3)
485   #define TPM_PT_YEAR                 (TPM_PT)(PT_FIXED+4)
486   #define TPM_PT_MANUFACTURER         (TPM_PT)(PT_FIXED+5)
487   #define TPM_PT_VENDOR_STRING_1      (TPM_PT)(PT_FIXED+6)
488   #define TPM_PT_VENDOR_STRING_2      (TPM_PT)(PT_FIXED+7)
489   #define TPM_PT_VENDOR_STRING_3      (TPM_PT)(PT_FIXED+8)
490   #define TPM_PT_VENDOR_STRING_4      (TPM_PT)(PT_FIXED+9)
491   #define TPM_PT_VENDOR_TPM_TYPE      (TPM_PT)(PT_FIXED+10)
492   #define TPM_PT_FIRMWARE_VERSION_1   (TPM_PT)(PT_FIXED+11)
493   #define TPM_PT_FIRMWARE_VERSION_2   (TPM_PT)(PT_FIXED+12)
494   #define TPM_PT_INPUT_BUFFER         (TPM_PT)(PT_FIXED+13)
495   #define TPM_PT_HR_TRANSIENT_MIN     (TPM_PT)(PT_FIXED+14)
496   #define TPM_PT_HR_PERSISTENT_MIN    (TPM_PT)(PT_FIXED+15)
497   #define TPM_PT_HR_LOADED_MIN        (TPM_PT)(PT_FIXED+16)
498   #define TPM_PT_ACTIVE_SESSIONS_MAX  (TPM_PT)(PT_FIXED+17)
499   #define TPM_PT_PCR_COUNT            (TPM_PT)(PT_FIXED+18)
500   #define TPM_PT_PCR_SELECT_MIN       (TPM_PT)(PT_FIXED+19)
501   #define TPM_PT_CONTEXT_GAP_MAX      (TPM_PT)(PT_FIXED+20)
502   #define TPM_PT_NV_COUNTERS_MAX      (TPM_PT)(PT_FIXED+22)
503   #define TPM_PT_NV_INDEX_MAX         (TPM_PT)(PT_FIXED+23)
504   #define TPM_PT_MEMORY               (TPM_PT)(PT_FIXED+24)
505   #define TPM_PT_CLOCK_UPDATE         (TPM_PT)(PT_FIXED+25)
506   #define TPM_PT_CONTEXT_HASH         (TPM_PT)(PT_FIXED+26)
507   #define TPM_PT_CONTEXT_SYM          (TPM_PT)(PT_FIXED+27)
508   #define TPM_PT_CONTEXT_SYM_SIZE     (TPM_PT)(PT_FIXED+28)
509   #define TPM_PT_ORDERLY_COUNT        (TPM_PT)(PT_FIXED+29)
510   #define TPM_PT_MAX_COMMAND_SIZE     (TPM_PT)(PT_FIXED+30)
511   #define TPM_PT_MAX_RESPONSE_SIZE    (TPM_PT)(PT_FIXED+31)
512   #define TPM_PT_MAX_DIGEST           (TPM_PT)(PT_FIXED+32)
513   #define TPM_PT_MAX_OBJECT_CONTEXT   (TPM_PT)(PT_FIXED+33)
514   #define TPM_PT_MAX_SESSION_CONTEXT  (TPM_PT)(PT_FIXED+34)
515   #define TPM_PT_PS_FAMILY_INDICATOR  (TPM_PT)(PT_FIXED+35)
516   #define TPM_PT_PS_LEVEL             (TPM_PT)(PT_FIXED+36)
517   #define TPM_PT_PS_REVISION          (TPM_PT)(PT_FIXED+37)
518   #define TPM_PT_PS_DAY_OF_YEAR       (TPM_PT)(PT_FIXED+38)
519   #define TPM_PT_PS_YEAR              (TPM_PT)(PT_FIXED+39)
520   #define TPM_PT_SPLIT_MAX            (TPM_PT)(PT_FIXED+40)
521   #define TPM_PT_TOTAL_COMMANDS       (TPM_PT)(PT_FIXED+41)
522   #define TPM_PT_LIBRARY_COMMANDS     (TPM_PT)(PT_FIXED+42)
523   #define TPM_PT_VENDOR_COMMANDS      (TPM_PT)(PT_FIXED+43)
524   #define TPM_PT_NV_BUFFER_MAX        (TPM_PT)(PT_FIXED+44)
525   #define TPM_PT_MODES                (TPM_PT)(PT_FIXED+45)
526   #define TPM_PT_MAX_CAP_BUFFER       (TPM_PT)(PT_FIXED+46)
527   #define PT_VAR                      (TPM_PT)(PT_GROUP*2)
528   #define TPM_PT_PERMANENT            (TPM_PT)(PT_VAR+0)
529   #define TPM_PT_STARTUP_CLEAR        (TPM_PT)(PT_VAR+1)
530   #define TPM_PT_HR_NV_INDEX          (TPM_PT)(PT_VAR+2)
531   #define TPM_PT_HR_LOADED            (TPM_PT)(PT_VAR+3)
532   #define TPM_PT_HR_LOADED_AVAIL      (TPM_PT)(PT_VAR+4)
533   #define TPM_PT_HR_ACTIVE            (TPM_PT)(PT_VAR+5)
534   #define TPM_PT_HR_ACTIVE_AVAIL      (TPM_PT)(PT_VAR+6)
535   #define TPM_PT_HR_TRANSIENT_AVAIL   (TPM_PT)(PT_VAR+7)
536   #define TPM_PT_HR_PERSISTENT        (TPM_PT)(PT_VAR+8)
537   #define TPM_PT_HR_PERSISTENT_AVAIL  (TPM_PT)(PT_VAR+9)
538   #define TPM_PT_NV_COUNTERS          (TPM_PT)(PT_VAR+10)
539   #define TPM_PT_NV_COUNTERS_AVAIL    (TPM_PT)(PT_VAR+11)
```



```
#define TPM_PT_ALGORITHM_SET        (TPM_PT)(PT_VAR+12)
#define TPM_PT_LOADED_CURVES        (TPM_PT)(PT_VAR+13)
#define TPM_PT_LOCKOUT_COUNTER      (TPM_PT)(PT_VAR+14)
#define TPM_PT_MAX_AUTH_FAIL        (TPM_PT)(PT_VAR+15)
#define TPM_PT_LOCKOUT_INTERVAL     (TPM_PT)(PT_VAR+16)
#define TPM_PT_LOCKOUT_RECOVERY     (TPM_PT)(PT_VAR+17)
#define TPM_PT_NV_WRITE_RECOVERY    (TPM_PT)(PT_VAR+18)
#define TPM_PT_AUDIT_COUNTER_0      (TPM_PT)(PT_VAR+19)
#define TPM_PT_AUDIT_COUNTER_1      (TPM_PT)(PT_VAR+20)
```

Table 2:24 - Definition of TPM_PT_PCR Constants

```
typedef UINT32                      TPM_PT_PCR;
#define TYPE_OF_TPM_PT_PCR          UINT32
#define TPM_PT_PCR_FIRST            (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_SAVE             (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_EXTEND_L0        (TPM_PT_PCR)(0x00000001)
#define TPM_PT_PCR_RESET_L0         (TPM_PT_PCR)(0x00000002)
#define TPM_PT_PCR_EXTEND_L1        (TPM_PT_PCR)(0x00000003)
#define TPM_PT_PCR_RESET_L1         (TPM_PT_PCR)(0x00000004)
#define TPM_PT_PCR_EXTEND_L2        (TPM_PT_PCR)(0x00000005)
#define TPM_PT_PCR_RESET_L2         (TPM_PT_PCR)(0x00000006)
#define TPM_PT_PCR_EXTEND_L3        (TPM_PT_PCR)(0x00000007)
#define TPM_PT_PCR_RESET_L3         (TPM_PT_PCR)(0x00000008)
#define TPM_PT_PCR_EXTEND_L4        (TPM_PT_PCR)(0x00000009)
#define TPM_PT_PCR_RESET_L4         (TPM_PT_PCR)(0x0000000A)
#define TPM_PT_PCR_NO_INCREMENT     (TPM_PT_PCR)(0x00000011)
#define TPM_PT_PCR_DRTM_RESET       (TPM_PT_PCR)(0x00000012)
#define TPM_PT_PCR_POLICY           (TPM_PT_PCR)(0x00000013)
#define TPM_PT_PCR_AUTH             (TPM_PT_PCR)(0x00000014)
#define TPM_PT_PCR_LAST             (TPM_PT_PCR)(0x00000014)
```

Table 2:25 - Definition of TPM_PS Constants

```
typedef UINT32                  TPM_PS;
#define TYPE_OF_TPM_PS          UINT32
#define TPM_PS_MAIN             (TPM_PS)(0x00000000)
#define TPM_PS_PC               (TPM_PS)(0x00000001)
#define TPM_PS_PDA              (TPM_PS)(0x00000002)
#define TPM_PS_CELL_PHONE       (TPM_PS)(0x00000003)
#define TPM_PS_SERVER           (TPM_PS)(0x00000004)
#define TPM_PS_PERIPHERAL       (TPM_PS)(0x00000005)
#define TPM_PS_TSS              (TPM_PS)(0x00000006)
#define TPM_PS_STORAGE          (TPM_PS)(0x00000007)
#define TPM_PS_AUTHENTICATION   (TPM_PS)(0x00000008)
#define TPM_PS_EMBEDDED         (TPM_PS)(0x00000009)
#define TPM_PS_HARDCOPY         (TPM_PS)(0x0000000A)
#define TPM_PS_INFRASTRUCTURE   (TPM_PS)(0x0000000B)
#define TPM_PS_VIRTUALIZATION   (TPM_PS)(0x0000000C)
#define TPM_PS_TNC              (TPM_PS)(0x0000000D)
#define TPM_PS_MULTI_TENANT     (TPM_PS)(0x0000000E)
#define TPM_PS_TC               (TPM_PS)(0x0000000F)
```

Table 2:26 - Definition of Types for Handles

```
typedef UINT32              TPM_HANDLE;
#define TYPE_OF_TPM_HANDLE  UINT32
```

Table 2:27 - Definition of TPM_HT Constants

```
typedef UINT8                   TPM_HT;
#define TYPE_OF_TPM_HT          UINT8
#define TPM_HT_PCR              (TPM_HT)(0x00)
#define TPM_HT_NV_INDEX         (TPM_HT)(0x01)
```

```
592  #define TPM_HT_HMAC_SESSION     (TPM_HT)(0x02)
593  #define TPM_HT_LOADED_SESSION   (TPM_HT)(0x02)
594  #define TPM_HT_POLICY_SESSION   (TPM_HT)(0x03)
595  #define TPM_HT_SAVED_SESSION    (TPM_HT)(0x03)
596  #define TPM_HT_PERMANENT        (TPM_HT)(0x40)
597  #define TPM_HT_TRANSIENT        (TPM_HT)(0x80)
598  #define TPM_HT_PERSISTENT       (TPM_HT)(0x81)
599  #define TPM_HT_AC               (TPM_HT)(0x90)
```

Table 2:28 - Definition of TPM_RH Constants

```
600  typedef TPM_HANDLE          TPM_RH;
601  #define TPM_RH_FIRST        (TPM_RH)(0x40000000)
602  #define TPM_RH_SRK          (TPM_RH)(0x40000000)
603  #define TPM_RH_OWNER        (TPM_RH)(0x40000001)
604  #define TPM_RH_REVOKE       (TPM_RH)(0x40000002)
605  #define TPM_RH_TRANSPORT    (TPM_RH)(0x40000003)
606  #define TPM_RH_OPERATOR     (TPM_RH)(0x40000004)
607  #define TPM_RH_ADMIN        (TPM_RH)(0x40000005)
608  #define TPM_RH_EK           (TPM_RH)(0x40000006)
609  #define TPM_RH_NULL         (TPM_RH)(0x40000007)
610  #define TPM_RH_UNASSIGNED   (TPM_RH)(0x40000008)
611  #define TPM_RS_PW           (TPM_RH)(0x40000009)
612  #define TPM_RH_LOCKOUT      (TPM_RH)(0x4000000A)
613  #define TPM_RH_ENDORSEMENT  (TPM_RH)(0x4000000B)
614  #define TPM_RH_PLATFORM     (TPM_RH)(0x4000000C)
615  #define TPM_RH_PLATFORM_NV  (TPM_RH)(0x4000000D)
616  #define TPM_RH_AUTH_00      (TPM_RH)(0x40000010)
617  #define TPM_RH_AUTH_FF      (TPM_RH)(0x4000010F)
618  #define TPM_RH_ACT_0        (TPM_RH)(0x40000110)
619  #define TPM_RH_ACT_F        (TPM_RH)(0x4000011F)
620  #define TPM_RH_LAST         (TPM_RH)(0x4000011F)
```

Table 2:29 - Definition of TPM_HC Constants

```
621  typedef TPM_HANDLE              TPM_HC;
622  #define HR_HANDLE_MASK          (TPM_HC)(0x00FFFFFF)
623  #define HR_RANGE_MASK           (TPM_HC)(0xFF000000)
624  #define HR_SHIFT                (TPM_HC)(24)
625  #define HR_PCR                  (TPM_HC)((TPM_HT_PCR<<HR_SHIFT))
626  #define HR_HMAC_SESSION         (TPM_HC)((TPM_HT_HMAC_SESSION<<HR_SHIFT))
627  #define HR_POLICY_SESSION       (TPM_HC)((TPM_HT_POLICY_SESSION<<HR_SHIFT))
628  #define HR_TRANSIENT            (TPM_HC)((TPM_HT_TRANSIENT<<HR_SHIFT))
629  #define HR_PERSISTENT           (TPM_HC)((TPM_HT_PERSISTENT<<HR_SHIFT))
630  #define HR_NV_INDEX             (TPM_HC)((TPM_HT_NV_INDEX<<HR_SHIFT))
631  #define HR_PERMANENT            (TPM_HC)((TPM_HT_PERMANENT<<HR_SHIFT))
632  #define PCR_FIRST               (TPM_HC)((HR_PCR+0))
633  #define PCR_LAST                (TPM_HC)((PCR_FIRST+IMPLEMENTATION_PCR-1))
634  #define HMAC_SESSION_FIRST      (TPM_HC)((HR_HMAC_SESSION+0))
635  #define HMAC_SESSION_LAST       (TPM_HC)((HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
636  #define LOADED_SESSION_FIRST    (TPM_HC)(HMAC_SESSION_FIRST)
637  #define LOADED_SESSION_LAST     (TPM_HC)(HMAC_SESSION_LAST)
638  #define POLICY_SESSION_FIRST    (TPM_HC)((HR_POLICY_SESSION+0))
639  #define POLICY_SESSION_LAST     \
640              (TPM_HC)((POLICY_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
641  #define TRANSIENT_FIRST         (TPM_HC)((HR_TRANSIENT+0))
642  #define ACTIVE_SESSION_FIRST    (TPM_HC)(POLICY_SESSION_FIRST)
643  #define ACTIVE_SESSION_LAST     (TPM_HC)(POLICY_SESSION_LAST)
644  #define TRANSIENT_LAST          (TPM_HC)((TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1))
645  #define PERSISTENT_FIRST        (TPM_HC)((HR_PERSISTENT+0))
646  #define PERSISTENT_LAST         (TPM_HC)((PERSISTENT_FIRST+0x00FFFFFF))
647  #define PLATFORM_PERSISTENT     (TPM_HC)((PERSISTENT_FIRST+0x00800000))
648  #define NV_INDEX_FIRST          (TPM_HC)((HR_NV_INDEX+0))
649  #define NV_INDEX_LAST           (TPM_HC)((NV_INDEX_FIRST+0x00FFFFFF))
650  #define PERMANENT_FIRST         (TPM_HC)(TPM_RH_FIRST)
```

```
651  #define PERMANENT_LAST          (TPM_HC)(TPM_RH_LAST)
652  #define HR_NV_AC                (TPM_HC)(((TPM_HT_NV_INDEX<<HR_SHIFT)+0xD00000))
653  #define NV_AC_FIRST             (TPM_HC)((HR_NV_AC+0))
654  #define NV_AC_LAST              (TPM_HC)((HR_NV_AC+0x0000FFFF))
655  #define HR_AC                   (TPM_HC)((TPM_HT_AC<<HR_SHIFT))
656  #define AC_FIRST                (TPM_HC)((HR_AC+0))
657  #define AC_LAST                 (TPM_HC)((HR_AC+0x0000FFFF))
658  #define TYPE_OF_TPMA_ALGORITHM  UINT32
659  #define TPMA_ALGORITHM_TO_UINT32(a)   (*((UINT32 *)&(a)))
660  #define UINT32_TO_TPMA_ALGORITHM(a)   (*((TPMA_ALGORITHM *)&(a)))
661  #define TPMA_ALGORITHM_TO_BYTE_ARRAY(i, a)                                  \
662              UINT32_TO_BYTE_ARRAY((TPMA_ALGORITHM_TO_UINT32(i)), (a))
663  #define BYTE_ARRAY_TO_TPMA_ALGORITHM(i, a)                                  \
664              {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                            \
665               i = UINT32_TO_TPMA_ALGORITHM(x);                               \
666               }
667  #if USE_BIT_FIELD_STRUCTURES
668  typedef struct TPMA_ALGORITHM {                     // Table 2:30
669      unsigned    asymmetric          : 1;
670      unsigned    symmetric           : 1;
671      unsigned    hash                : 1;
672      unsigned    object              : 1;
673      unsigned    Reserved_bits_at_4  : 4;
674      unsigned    signing             : 1;
675      unsigned    encrypting          : 1;
676      unsigned    method              : 1;
677      unsigned    Reserved_bits_at_11 : 21;
678  } TPMA_ALGORITHM;                                   /* Bits */
```

This is the initializer for a TPMA_ALGORITHM structure

```
679  #define TPMA_ALGORITHM_INITIALIZER(                                        \
680               asymmetric, symmetric,  hash,       object,     bits_at_4,    \
681               signing,    encrypting, method,     bits_at_11)               \
682              {asymmetric, symmetric,  hash,       object,     bits_at_4,    \
683               signing,    encrypting, method,     bits_at_11}
684  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:30 TPMA_ALGORITHM using bit masking

```
685  typedef UINT32                      TPMA_ALGORITHM;
686  #define TYPE_OF_TPMA_ALGORITHM      UINT32
687  #define TPMA_ALGORITHM_asymmetric   ((TPMA_ALGORITHM)1 << 0)
688  #define TPMA_ALGORITHM_symmetric    ((TPMA_ALGORITHM)1 << 1)
689  #define TPMA_ALGORITHM_hash         ((TPMA_ALGORITHM)1 << 2)
690  #define TPMA_ALGORITHM_object       ((TPMA_ALGORITHM)1 << 3)
691  #define TPMA_ALGORITHM_signing      ((TPMA_ALGORITHM)1 << 8)
692  #define TPMA_ALGORITHM_encrypting   ((TPMA_ALGORITHM)1 << 9)
693  #define TPMA_ALGORITHM_method       ((TPMA_ALGORITHM)1 << 10)
```

This is the initializer for a TPMA_ALGORITHM bit array.

```
694  #define TPMA_ALGORITHM_INITIALIZER(                                        \
695               asymmetric, symmetric,  hash,       object,     bits_at_4,    \
696               signing,    encrypting, method,     bits_at_11)               \
697              {(asymmetric << 0) + (symmetric << 1)  + (hash << 2)       +    \
698               (object << 3)     + (signing << 8)    + (encrypting << 9) +    \
699               (method << 10)}
700  #endif // USE_BIT_FIELD_STRUCTURES
701  #define TYPE_OF_TPMA_OBJECT UINT32
702  #define TPMA_OBJECT_TO_UINT32(a)     (*((UINT32 *)&(a)))
703  #define UINT32_TO_TPMA_OBJECT(a)     (*((TPMA_OBJECT *)&(a)))
704  #define TPMA_OBJECT_TO_BYTE_ARRAY(i, a)                                    \
705              UINT32_TO_BYTE_ARRAY((TPMA_OBJECT_TO_UINT32(i)), (a))
```

```
706  #define BYTE_ARRAY_TO_TPMA_OBJECT(i, a)                                          \
707              { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_OBJECT(x); }
708  #if USE_BIT_FIELD_STRUCTURES
709  typedef struct TPMA_OBJECT {                          // Table 2:31
710      unsigned    Reserved_bit_at_0        : 1;
711      unsigned    fixedTPM                 : 1;
712      unsigned    stClear                  : 1;
713      unsigned    Reserved_bit_at_3        : 1;
714      unsigned    fixedParent              : 1;
715      unsigned    sensitiveDataOrigin      : 1;
716      unsigned    userWithAuth             : 1;
717      unsigned    adminWithPolicy          : 1;
718      unsigned    Reserved_bits_at_8       : 2;
719      unsigned    noDA                     : 1;
720      unsigned    encryptedDuplication     : 1;
721      unsigned    Reserved_bits_at_12      : 4;
722      unsigned    restricted               : 1;
723      unsigned    decrypt                  : 1;
724      unsigned    sign                     : 1;
725      unsigned    x509sign                 : 1;
726      unsigned    Reserved_bits_at_20      : 12;
727  } TPMA_OBJECT;                                        /* Bits */
```

This is the initializer for a TPMA_OBJECT structure

```
728  #define TPMA_OBJECT_INITIALIZER(                                                 \
729              bit_at_0,             fixedtpm,             stclear,                \
730              bit_at_3,             fixedparent,          sensitivedataorigin,    \
731              userwithauth,         adminwithpolicy,      bits_at_8,              \
732              noda,                 encryptedduplication, bits_at_12,             \
733              restricted,           decrypt,              sign,                   \
734              x509sign,             bits_at_20)                                   \
735             {bit_at_0,             fixedtpm,             stclear,                \
736              bit_at_3,             fixedparent,          sensitivedataorigin,    \
737              userwithauth,         adminwithpolicy,      bits_at_8,              \
738              noda,                 encryptedduplication, bits_at_12,             \
739              restricted,           decrypt,              sign,                   \
740              x509sign,             bits_at_20}
741  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:31 TPMA_OBJECT using bit masking

```
742  typedef UINT32                          TPMA_OBJECT;
743  #define TYPE_OF_TPMA_OBJECT             UINT32
744  #define TPMA_OBJECT_fixedTPM            ((TPMA_OBJECT)1 << 1)
745  #define TPMA_OBJECT_stClear             ((TPMA_OBJECT)1 << 2)
746  #define TPMA_OBJECT_fixedParent         ((TPMA_OBJECT)1 << 4)
747  #define TPMA_OBJECT_sensitiveDataOrigin ((TPMA_OBJECT)1 << 5)
748  #define TPMA_OBJECT_userWithAuth        ((TPMA_OBJECT)1 << 6)
749  #define TPMA_OBJECT_adminWithPolicy     ((TPMA_OBJECT)1 << 7)
750  #define TPMA_OBJECT_noDA                ((TPMA_OBJECT)1 << 10)
751  #define TPMA_OBJECT_encryptedDuplication ((TPMA_OBJECT)1 << 11)
752  #define TPMA_OBJECT_restricted          ((TPMA_OBJECT)1 << 16)
753  #define TPMA_OBJECT_decrypt             ((TPMA_OBJECT)1 << 17)
754  #define TPMA_OBJECT_sign                ((TPMA_OBJECT)1 << 18)
755  #define TPMA_OBJECT_x509sign            ((TPMA_OBJECT)1 << 19)
```

This is the initializer for a TPMA_OBJECT bit array.

```
756  #define TPMA_OBJECT_INITIALIZER(                                                 \
757              bit_at_0,             fixedtpm,             stclear,                \
758              bit_at_3,             fixedparent,          sensitivedataorigin,    \
759              userwithauth,         adminwithpolicy,      bits_at_8,              \
760              noda,                 encryptedduplication, bits_at_12,             \
```

```
761                 restricted,           decrypt,              sign,                   \
762                 x509sign,             bits_at_20)                                   \
763                {(fixedtpm << 1)              + (stclear << 2)               +       \
764                 (fixedparent << 4)           + (sensitivedataorigin << 5)   +       \
765                 (userwithauth << 6)          + (adminwithpolicy << 7)       +       \
766                 (noda << 10)                 + (encryptedduplication << 11) +       \
767                 (restricted << 16)           + (decrypt << 17)              +       \
768                 (sign << 18)                 + (x509sign << 19)}
769    #endif // USE_BIT_FIELD_STRUCTURES
770    #define TYPE_OF_TPMA_SESSION    UINT8
771    #define TPMA_SESSION_TO_UINT8(a)     (*((UINT8 *)&(a)))
772    #define UINT8_TO_TPMA_SESSION(a)     (*((TPMA_SESSION *)&(a)))
773    #define TPMA_SESSION_TO_BYTE_ARRAY(i, a)                                          \
774                UINT8_TO_BYTE_ARRAY((TPMA_SESSION_TO_UINT8(i)), (a))
775    #define BYTE_ARRAY_TO_TPMA_SESSION(i, a)                                          \
776                { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_SESSION(x); }
777    #if USE_BIT_FIELD_STRUCTURES
778    typedef struct TPMA_SESSION {                        // Table 2:32
779        unsigned    continueSession      : 1;
780        unsigned    auditExclusive       : 1;
781        unsigned    auditReset           : 1;
782        unsigned    Reserved_bits_at_3   : 2;
783        unsigned    decrypt              : 1;
784        unsigned    encrypt              : 1;
785        unsigned    audit                : 1;
786    } TPMA_SESSION;                                      /* Bits */
```

This is the initializer for a TPMA_SESSION structure

```
787    #define TPMA_SESSION_INITIALIZER(                                                 \
788                continuesession, auditexclusive,  auditreset,      bits_at_3,         \
789                decrypt,         encrypt,         audit)                             \
790               {continuesession, auditexclusive,  auditreset,      bits_at_3,         \
791                decrypt,         encrypt,         audit}
792    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:32 TPMA_SESSION using bit masking

```
793    typedef UINT8                          TPMA_SESSION;
794    #define TYPE_OF_TPMA_SESSION           UINT8
795    #define TPMA_SESSION_continueSession   ((TPMA_SESSION)1 << 0)
796    #define TPMA_SESSION_auditExclusive    ((TPMA_SESSION)1 << 1)
797    #define TPMA_SESSION_auditReset        ((TPMA_SESSION)1 << 2)
798    #define TPMA_SESSION_decrypt           ((TPMA_SESSION)1 << 5)
799    #define TPMA_SESSION_encrypt           ((TPMA_SESSION)1 << 6)
800    #define TPMA_SESSION_audit             ((TPMA_SESSION)1 << 7)
```

This is the initializer for a TPMA_SESSION bit array.

```
801    #define TPMA_SESSION_INITIALIZER(                                                 \
802                continuesession, auditexclusive,  auditreset,      bits_at_3,         \
803                decrypt,         encrypt,         audit)                             \
804               {(continuesession << 0) + (auditexclusive << 1)  +                     \
805                (auditreset << 2)      + (decrypt << 5)         +                     \
806                (encrypt << 6)         + (audit << 7)}
807    #endif // USE_BIT_FIELD_STRUCTURES
808    #define TYPE_OF_TPMA_LOCALITY   UINT8
809    #define TPMA_LOCALITY_TO_UINT8(a)     (*((UINT8 *)&(a)))
810    #define UINT8_TO_TPMA_LOCALITY(a)     (*((TPMA_LOCALITY *)&(a)))
811    #define TPMA_LOCALITY_TO_BYTE_ARRAY(i, a)                                         \
812                UINT8_TO_BYTE_ARRAY((TPMA_LOCALITY_TO_UINT8(i)), (a))
813    #define BYTE_ARRAY_TO_TPMA_LOCALITY(i, a)                                         \
814                { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_LOCALITY(x); }
815    #if USE_BIT_FIELD_STRUCTURES
```

```
816  typedef struct TPMA_LOCALITY {                        // Table 2:33
817      unsigned    TPM_LOC_ZERO          : 1;
818      unsigned    TPM_LOC_ONE           : 1;
819      unsigned    TPM_LOC_TWO           : 1;
820      unsigned    TPM_LOC_THREE         : 1;
821      unsigned    TPM_LOC_FOUR          : 1;
822      unsigned    Extended              : 3;
823  } TPMA_LOCALITY;                                      /* Bits */
```

This is the initializer for a TPMA_LOCALITY structure

```
824  #define TPMA_LOCALITY_INITIALIZER(                                                \
825              tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,           \
826              tpm_loc_four,  extended)                                              \
827             {tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,           \
828              tpm_loc_four,  extended}
829  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:33 TPMA_LOCALITY using bit masking

```
830  typedef UINT8                           TPMA_LOCALITY;
831  #define TYPE_OF_TPMA_LOCALITY           UINT8
832  #define TPMA_LOCALITY_TPM_LOC_ZERO      ((TPMA_LOCALITY)1 << 0)
833  #define TPMA_LOCALITY_TPM_LOC_ONE       ((TPMA_LOCALITY)1 << 1)
834  #define TPMA_LOCALITY_TPM_LOC_TWO       ((TPMA_LOCALITY)1 << 2)
835  #define TPMA_LOCALITY_TPM_LOC_THREE     ((TPMA_LOCALITY)1 << 3)
836  #define TPMA_LOCALITY_TPM_LOC_FOUR      ((TPMA_LOCALITY)1 << 4)
837  #define TPMA_LOCALITY_Extended_SHIFT    5
838  #define TPMA_LOCALITY_Extended          ((TPMA_LOCALITY)0x7 << 5)
```

This is the initializer for a TPMA_LOCALITY bit array.

```
839  #define TPMA_LOCALITY_INITIALIZER(                                                \
840              tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,           \
841              tpm_loc_four,  extended)                                              \
842             {(tpm_loc_zero << 0)  + (tpm_loc_one << 1)   + (tpm_loc_two << 2)   +  \
843              (tpm_loc_three << 3) + (tpm_loc_four << 4)  + (extended << 5)}
844  #endif // USE_BIT_FIELD_STRUCTURES
845  #define TYPE_OF_TPMA_PERMANENT  UINT32
846  #define TPMA_PERMANENT_TO_UINT32(a)   (*((UINT32 *)&(a)))
847  #define UINT32_TO_TPMA_PERMANENT(a)   (*((TPMA_PERMANENT *)&(a)))
848  #define TPMA_PERMANENT_TO_BYTE_ARRAY(i, a)                                        \
849              UINT32_TO_BYTE_ARRAY((TPMA_PERMANENT_TO_UINT32(i)), (a))
850  #define BYTE_ARRAY_TO_TPMA_PERMANENT(i, a)                                        \
851              {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                                  \
852               i = UINT32_TO_TPMA_PERMANENT(x);                                     \
853               }
854  #if USE_BIT_FIELD_STRUCTURES
855  typedef struct TPMA_PERMANENT {                       // Table 2:34
856      unsigned    ownerAuthSet          : 1;
857      unsigned    endorsementAuthSet    : 1;
858      unsigned    lockoutAuthSet        : 1;
859      unsigned    Reserved_bits_at_3    : 5;
860      unsigned    disableClear          : 1;
861      unsigned    inLockout             : 1;
862      unsigned    tpmGeneratedEPS       : 1;
863      unsigned    Reserved_bits_at_11   : 21;
864  } TPMA_PERMANENT;                                     /* Bits */
```

This is the initializer for a TPMA_PERMANENT structure

```
865  #define TPMA_PERMANENT_INITIALIZER(                                               \
866              ownerauthset,       endorsementauthset, lockoutauthset,           \
867              bits_at_3,          disableclear,       inlockout,                \
```

```
868                 tpmgeneratedeps,    bits_at_11)                                 \
869               {ownerauthset,        endorsementauthset, lockoutauthset,         \
870                 bits_at_3,          disableclear,       inlockout,              \
871                 tpmgeneratedeps,    bits_at_11}
872     #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:34 TPMA_PERMANENT using bit masking

```
873     typedef UINT32                              TPMA_PERMANENT;
874     #define TYPE_OF_TPMA_PERMANENT              UINT32
875     #define TPMA_PERMANENT_ownerAuthSet         ((TPMA_PERMANENT)1 << 0)
876     #define TPMA_PERMANENT_endorsementAuthSet   ((TPMA_PERMANENT)1 << 1)
877     #define TPMA_PERMANENT_lockoutAuthSet       ((TPMA_PERMANENT)1 << 2)
878     #define TPMA_PERMANENT_disableClear         ((TPMA_PERMANENT)1 << 8)
879     #define TPMA_PERMANENT_inLockout            ((TPMA_PERMANENT)1 << 9)
880     #define TPMA_PERMANENT_tpmGeneratedEPS      ((TPMA_PERMANENT)1 << 10)
```

This is the initializer for a TPMA_PERMANENT bit array.

```
881     #define TPMA_PERMANENT_INITIALIZER(                                         \
882                 ownerauthset,       endorsementauthset, lockoutauthset,         \
883                 bits_at_3,          disableclear,       inlockout,              \
884                 tpmgeneratedeps,    bits_at_11)                                 \
885               {(ownerauthset << 0)       + (endorsementauthset << 1) +          \
886                (lockoutauthset << 2)     + (disableclear << 8)       +          \
887                (inlockout << 9)          + (tpmgeneratedeps << 10)}
888     #endif // USE_BIT_FIELD_STRUCTURES
889     #define TYPE_OF_TPMA_STARTUP_CLEAR  UINT32
890     #define TPMA_STARTUP_CLEAR_TO_UINT32(a)   (*((UINT32 *)&(a)))
891     #define UINT32_TO_TPMA_STARTUP_CLEAR(a)   (*((TPMA_STARTUP_CLEAR *)&(a)))
892     #define TPMA_STARTUP_CLEAR_TO_BYTE_ARRAY(i, a)                              \
893             UINT32_TO_BYTE_ARRAY((TPMA_STARTUP_CLEAR_TO_UINT32(i)), (a))
894     #define BYTE_ARRAY_TO_TPMA_STARTUP_CLEAR(i, a)                              \
895             {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                                \
896              i = UINT32_TO_TPMA_STARTUP_CLEAR(x);                               \
897              }
898     #if USE_BIT_FIELD_STRUCTURES
899     typedef struct TPMA_STARTUP_CLEAR {                 // Table 2:35
900         unsigned    phEnable            : 1;
901         unsigned    shEnable            : 1;
902         unsigned    ehEnable            : 1;
903         unsigned    phEnableNV          : 1;
904         unsigned    Reserved_bits_at_4  : 27;
905         unsigned    orderly             : 1;
906     } TPMA_STARTUP_CLEAR;                           /* Bits */
```

This is the initializer for a TPMA_STARTUP_CLEAR structure

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

This implements Table 2:35 TPMA_STARTUP_CLEAR using bit masking

```
911     typedef UINT32                          TPMA_STARTUP_CLEAR;
912     #define TYPE_OF_TPMA_STARTUP_CLEAR      UINT32
913     #define TPMA_STARTUP_CLEAR_phEnable     ((TPMA_STARTUP_CLEAR)1 << 0)
914     #define TPMA_STARTUP_CLEAR_shEnable     ((TPMA_STARTUP_CLEAR)1 << 1)
915     #define TPMA_STARTUP_CLEAR_ehEnable     ((TPMA_STARTUP_CLEAR)1 << 2)
916     #define TPMA_STARTUP_CLEAR_phEnableNV   ((TPMA_STARTUP_CLEAR)1 << 3)
917     #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
#define TYPE_OF_TPMA_MEMORY UINT32
#define TPMA_MEMORY_TO_UINT32(a)      (*((UINT32 *)&(a)))
#define UINT32_TO_TPMA_MEMORY(a)      (*((TPMA_MEMORY *)&(a)))
#define TPMA_MEMORY_TO_BYTE_ARRAY(i, a)                                                 \
            UINT32_TO_BYTE_ARRAY((TPMA_MEMORY_TO_UINT32(i)), (a))
#define BYTE_ARRAY_TO_TPMA_MEMORY(i, a)                                                 \
            { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MEMORY(x); }
#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 TYPE_OF_TPMA_MEMORY             UINT32
#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
#define TYPE_OF_TPMA_CC     UINT32
#define TPMA_CC_TO_UINT32(a)      (*((UINT32 *)&(a)))
#define UINT32_TO_TPMA_CC(a)      (*((TPMA_CC *)&(a)))
#define TPMA_CC_TO_BYTE_ARRAY(i, a)                                                     \
            UINT32_TO_BYTE_ARRAY((TPMA_CC_TO_UINT32(i)), (a))
#define BYTE_ARRAY_TO_TPMA_CC(i, a)                                                     \
            { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_CC(x); }
#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

```
969  #define TPMA_CC_INITIALIZER(                                                    \
970                  commandindex, bits_at_16,   nv,           extensive,    flushed,      \
971                  chandles,     rhandle,      v,            bits_at_30)                 \
972                 {commandindex, bits_at_16,   nv,           extensive,    flushed,      \
973                  chandles,     rhandle,      v,            bits_at_30}
974  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:37 TPMA_CC using bit masking

```
975  typedef UINT32                      TPMA_CC;
976  #define TYPE_OF_TPMA_CC             UINT32
977  #define TPMA_CC_commandIndex_SHIFT  0
978  #define TPMA_CC_commandIndex        ((TPMA_CC)0xffff << 0)
979  #define TPMA_CC_nv                  ((TPMA_CC)1 << 22)
980  #define TPMA_CC_extensive           ((TPMA_CC)1 << 23)
981  #define TPMA_CC_flushed             ((TPMA_CC)1 << 24)
982  #define TPMA_CC_cHandles_SHIFT      25
983  #define TPMA_CC_cHandles            ((TPMA_CC)0x7 << 25)
984  #define TPMA_CC_rHandle             ((TPMA_CC)1 << 28)
985  #define TPMA_CC_V                   ((TPMA_CC)1 << 29)
```

This is the initializer for a TPMA_CC bit array.

```
986  #define TPMA_CC_INITIALIZER(                                                    \
987                  commandindex, bits_at_16,   nv,           extensive,    flushed,      \
988                  chandles,     rhandle,      v,            bits_at_30)                 \
989                 {(commandindex << 0) + (nv << 22)          + (extensive << 23)   +     \
990                  (flushed << 24)     + (chandles << 25)    + (rhandle << 28)     +     \
991                  (v << 29)}
992  #endif // USE_BIT_FIELD_STRUCTURES
993  #define TYPE_OF_TPMA_MODES  UINT32
994  #define TPMA_MODES_TO_UINT32(a)   (*((UINT32 *)&(a)))
995  #define UINT32_TO_TPMA_MODES(a)   (*((TPMA_MODES *)&(a)))
996  #define TPMA_MODES_TO_BYTE_ARRAY(i, a)                                          \
997              UINT32_TO_BYTE_ARRAY((TPMA_MODES_TO_UINT32(i)), (a))
998  #define BYTE_ARRAY_TO_TPMA_MODES(i, a)                                          \
999              { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MODES(x); }
1000 #if USE_BIT_FIELD_STRUCTURES
1001 typedef struct TPMA_MODES {                         // Table 2:38
1002     unsigned    FIPS_140_2          : 1;
1003     unsigned    Reserved_bits_at_1  : 31;
1004 } TPMA_MODES;                                       /* Bits */
```

This is the initializer for a TPMA_MODES structure

```
1005 #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {fips_140_2, bits_at_1}
1006 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:38 TPMA_MODES using bit masking

```
1007 typedef UINT32                  TPMA_MODES;
1008 #define TYPE_OF_TPMA_MODES      UINT32
1009 #define TPMA_MODES_FIPS_140_2   ((TPMA_MODES)1 << 0)
```

This is the initializer for a TPMA_MODES bit array.

```
1010 #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {(fips_140_2 << 0)}
1011 #endif // USE_BIT_FIELD_STRUCTURES
1012 #define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
1013 #define TPMA_X509_KEY_USAGE_TO_UINT32(a)      (*((UINT32 *)&(a)))
1014 #define UINT32_TO_TPMA_X509_KEY_USAGE(a)      (*((TPMA_X509_KEY_USAGE *)&(a)))
1015 #define TPMA_X509_KEY_USAGE_TO_BYTE_ARRAY(i, a)                                 \
1016             UINT32_TO_BYTE_ARRAY((TPMA_X509_KEY_USAGE_TO_UINT32(i)), (a))
```

```
1017  #define BYTE_ARRAY_TO_TPMA_X509_KEY_USAGE(i, a)                                  \
1018              {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                                 \
1019               i = UINT32_TO_TPMA_X509_KEY_USAGE(x);                               \
1020               }
1021  #if USE_BIT_FIELD_STRUCTURES
1022  typedef struct TPMA_X509_KEY_USAGE {                  // Table 2:39
1023      unsigned    Reserved_bits_at_0   : 23;
1024      unsigned    decipherOnly         : 1;
1025      unsigned    encipherOnly         : 1;
1026      unsigned    cRLSign              : 1;
1027      unsigned    keyCertSign          : 1;
1028      unsigned    keyAgreement         : 1;
1029      unsigned    dataEncipherment     : 1;
1030      unsigned    keyEncipherment      : 1;
1031      unsigned    nonrepudiation       : 1;
1032      unsigned    digitalSignature     : 1;
1033  } TPMA_X509_KEY_USAGE;                                /* Bits */
```

This is the initializer for a TPMA_X509_KEY_USAGE structure

```
1034  #define TPMA_X509_KEY_USAGE_INITIALIZER(                                         \
1035              bits_at_0,        decipheronly,     encipheronly,                    \
1036              crlsign,          keycertsign,      keyagreement,                    \
1037              dataencipherment, keyencipherment,  nonrepudiation,                  \
1038              digitalsignature)                                                    \
1039             {bits_at_0,        decipheronly,     encipheronly,                    \
1040              crlsign,          keycertsign,      keyagreement,                    \
1041              dataencipherment, keyencipherment,  nonrepudiation,                  \
1042              digitalsignature}
1043  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:39 TPMA_X509_KEY_USAGE using bit masking

```
1044  typedef UINT32                                TPMA_X509_KEY_USAGE;
1045  #define TYPE_OF_TPMA_X509_KEY_USAGE           UINT32
1046  #define TPMA_X509_KEY_USAGE_decipherOnly      ((TPMA_X509_KEY_USAGE)1 << 23)
1047  #define TPMA_X509_KEY_USAGE_encipherOnly      ((TPMA_X509_KEY_USAGE)1 << 24)
1048  #define TPMA_X509_KEY_USAGE_cRLSign           ((TPMA_X509_KEY_USAGE)1 << 25)
1049  #define TPMA_X509_KEY_USAGE_keyCertSign       ((TPMA_X509_KEY_USAGE)1 << 26)
1050  #define TPMA_X509_KEY_USAGE_keyAgreement      ((TPMA_X509_KEY_USAGE)1 << 27)
1051  #define TPMA_X509_KEY_USAGE_dataEncipherment  ((TPMA_X509_KEY_USAGE)1 << 28)
1052  #define TPMA_X509_KEY_USAGE_keyEncipherment   ((TPMA_X509_KEY_USAGE)1 << 29)
1053  #define TPMA_X509_KEY_USAGE_nonrepudiation    ((TPMA_X509_KEY_USAGE)1 << 30)
1054  #define TPMA_X509_KEY_USAGE_digitalSignature  ((TPMA_X509_KEY_USAGE)1 << 31)
```

This is the initializer for a TPMA_X509_KEY_USAGE bit array.

```
1055  #define TPMA_X509_KEY_USAGE_INITIALIZER(                                         \
1056              bits_at_0,        decipheronly,     encipheronly,                    \
1057              crlsign,          keycertsign,      keyagreement,                    \
1058              dataencipherment, keyencipherment,  nonrepudiation,                  \
1059              digitalsignature)                                                    \
1060             {(decipheronly << 23)     + (encipheronly << 24)     +               \
1061              (crlsign << 25)          + (keycertsign << 26)      +               \
1062              (keyagreement << 27)     + (dataencipherment << 28) +               \
1063              (keyencipherment << 29)  + (nonrepudiation << 30)   +               \
1064              (digitalsignature << 31)}
1065  #endif // USE_BIT_FIELD_STRUCTURES
1066  #define TYPE_OF_TPMA_ACT    UINT32
1067  #define TPMA_ACT_TO_UINT32(a)     (*((UINT32 *)&(a)))
1068  #define UINT32_TO_TPMA_ACT(a)     (*((TPMA_ACT *)&(a)))
1069  #define TPMA_ACT_TO_BYTE_ARRAY(i, a)                                             \
1070              UINT32_TO_BYTE_ARRAY((TPMA_ACT_TO_UINT32(i)), (a))
1071  #define BYTE_ARRAY_TO_TPMA_ACT(i, a)                                             \
```

```
1072            { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_ACT(x); }
1073    #if USE_BIT_FIELD_STRUCTURES
1074    typedef struct TPMA_ACT {                           // Table 2:40
1075        unsigned    signaled            : 1;
1076        unsigned    preserveSignaled    : 1;
1077        unsigned    Reserved_bits_at_2  : 30;
1078    } TPMA_ACT;                                         /* Bits */
```

This is the initializer for a TPMA_ACT structure

```
1079    #define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2)             \
1080            {signaled, preservesignaled, bits_at_2}
1081    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:40 TPMA_ACT using bit masking

```
1082    typedef UINT32                      TPMA_ACT;
1083    #define TYPE_OF_TPMA_ACT            UINT32
1084    #define TPMA_ACT_signaled           ((TPMA_ACT)1 << 0)
1085    #define TPMA_ACT_preserveSignaled   ((TPMA_ACT)1 << 1)
```

This is the initializer for a TPMA_ACT bit array.

```
1086    #define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2)             \
1087            {(signaled << 0) + (preservesignaled << 1)}
1088    #endif // USE_BIT_FIELD_STRUCTURES
1089    typedef BYTE                TPMI_YES_NO;            // Table 2:41  /* Interface */
1090    typedef TPM_HANDLE          TPMI_DH_OBJECT;         // Table 2:42  /* Interface */
1091    typedef TPM_HANDLE          TPMI_DH_PARENT;         // Table 2:43  /* Interface */
1092    typedef TPM_HANDLE          TPMI_DH_PERSISTENT;     // Table 2:44  /* Interface */
1093    typedef TPM_HANDLE          TPMI_DH_ENTITY;         // Table 2:45  /* Interface */
1094    typedef TPM_HANDLE          TPMI_DH_PCR;            // Table 2:46  /* Interface */
1095    typedef TPM_HANDLE          TPMI_SH_AUTH_SESSION;   // Table 2:47  /* Interface */
1096    typedef TPM_HANDLE          TPMI_SH_HMAC;           // Table 2:48  /* Interface */
1097    typedef TPM_HANDLE          TPMI_SH_POLICY;         // Table 2:49  /* Interface */
1098    typedef TPM_HANDLE          TPMI_DH_CONTEXT;        // Table 2:50  /* Interface */
1099    typedef TPM_HANDLE          TPMI_DH_SAVED;          // Table 2:51  /* Interface */
1100    typedef TPM_HANDLE          TPMI_RH_HIERARCHY;      // Table 2:52  /* Interface */
1101    typedef TPM_HANDLE          TPMI_RH_ENABLES;        // Table 2:53  /* Interface */
1102    typedef TPM_HANDLE          TPMI_RH_HIERARCHY_AUTH; // Table 2:54  /* Interface */
1103    typedef TPM_HANDLE          TPMI_RH_HIERARCHY_POLICY;
1104    typedef TPM_HANDLE          TPMI_RH_PLATFORM;       // Table 2:56  /* Interface */
1105    typedef TPM_HANDLE          TPMI_RH_OWNER;          // Table 2:57  /* Interface */
1106    typedef TPM_HANDLE          TPMI_RH_ENDORSEMENT;    // Table 2:58  /* Interface */
1107    typedef TPM_HANDLE          TPMI_RH_PROVISION;      // Table 2:59  /* Interface */
1108    typedef TPM_HANDLE          TPMI_RH_CLEAR;          // Table 2:60  /* Interface */
1109    typedef TPM_HANDLE          TPMI_RH_NV_AUTH;        // Table 2:61  /* Interface */
1110    typedef TPM_HANDLE          TPMI_RH_LOCKOUT;        // Table 2:62  /* Interface */
1111    typedef TPM_HANDLE          TPMI_RH_NV_INDEX;       // Table 2:63  /* Interface */
1112    typedef TPM_HANDLE          TPMI_RH_AC;             // Table 2:64  /* Interface */
1113    typedef TPM_HANDLE          TPMI_RH_ACT;            // Table 2:65  /* Interface */
1114    typedef TPM_ALG_ID          TPMI_ALG_HASH;          // Table 2:66  /* Interface */
1115    typedef TPM_ALG_ID          TPMI_ALG_ASYM;          // Table 2:67  /* Interface */
1116    typedef TPM_ALG_ID          TPMI_ALG_SYM;           // Table 2:68  /* Interface */
1117    typedef TPM_ALG_ID          TPMI_ALG_SYM_OBJECT;    // Table 2:69  /* Interface */
1118    typedef TPM_ALG_ID          TPMI_ALG_SYM_MODE;      // Table 2:70  /* Interface */
1119    typedef TPM_ALG_ID          TPMI_ALG_KDF;           // Table 2:71  /* Interface */
1120    typedef TPM_ALG_ID          TPMI_ALG_SIG_SCHEME;    // Table 2:72  /* Interface */
1121    typedef TPM_ALG_ID          TPMI_ECC_KEY_EXCHANGE;  // Table 2:73  /* Interface */
1122    typedef TPM_ST              TPMI_ST_COMMAND_TAG;    // Table 2:74  /* Interface */
1123    typedef TPM_ALG_ID          TPMI_ALG_MAC_SCHEME;    // Table 2:75  /* Interface */
1124    typedef TPM_ALG_ID          TPMI_ALG_CIPHER_MODE;   // Table 2:76  /* Interface */
1125    typedef BYTE                TPMS_EMPTY;             // Table 2:77
1126    typedef struct {                                    // Table 2:78
```

```
1127        TPM_ALG_ID              alg;
1128        TPMA_ALGORITHM          attributes;
1129    } TPMS_ALGORITHM_DESCRIPTION;                       /* Structure */
1130    typedef union {                                     // Table 2:79
1131    #if ALG_SHA1
1132        BYTE                    sha1[SHA1_DIGEST_SIZE];
1133    #endif // ALG_SHA1
1134    #if ALG_SHA256
1135        BYTE                    sha256[SHA256_DIGEST_SIZE];
1136    #endif // ALG_SHA256
1137    #if ALG_SHA384
1138        BYTE                    sha384[SHA384_DIGEST_SIZE];
1139    #endif // ALG_SHA384
1140    #if ALG_SHA512
1141        BYTE                    sha512[SHA512_DIGEST_SIZE];
1142    #endif // ALG_SHA512
1143    #if ALG_SM3_256
1144        BYTE                    sm3_256[SM3_256_DIGEST_SIZE];
1145    #endif // ALG_SM3_256
1146    #if ALG_SHA3_256
1147        BYTE                    sha3_256[SHA3_256_DIGEST_SIZE];
1148    #endif // ALG_SHA3_256
1149    #if ALG_SHA3_384
1150        BYTE                    sha3_384[SHA3_384_DIGEST_SIZE];
1151    #endif // ALG_SHA3_384
1152    #if ALG_SHA3_512
1153        BYTE                    sha3_512[SHA3_512_DIGEST_SIZE];
1154    #endif // ALG_SHA3_512
1155    } TPMU_HA;                                          /* Structure */
1156    typedef struct {                                    // Table 2:80
1157        TPMI_ALG_HASH           hashAlg;
1158        TPMU_HA                 digest;
1159    } TPMT_HA;                                          /* Structure */
1160    typedef union {                                     // Table 2:81
1161        struct {
1162            UINT16              size;
1163            BYTE                buffer[sizeof(TPMU_HA)];
1164        }           t;
1165        TPM2B       b;
1166    } TPM2B_DIGEST;                                     /* Structure */
1167    typedef union {                                     // Table 2:82
1168        struct {
1169            UINT16              size;
1170            BYTE                buffer[sizeof(TPMT_HA)];
1171        }           t;
1172        TPM2B       b;
1173    } TPM2B_DATA;                                       /* Structure */
```

Table 2:83 - Definition of Types for TPM2B_NONCE

```
1174    typedef TPM2B_DIGEST        TPM2B_NONCE;
```

Table 2:84 - Definition of Types for TPM2B_AUTH

```
1175    typedef TPM2B_DIGEST        TPM2B_AUTH;
```

Table 2:85 - Definition of Types for TPM2B_OPERAND

```
1176    typedef TPM2B_DIGEST        TPM2B_OPERAND;
1177    typedef union {                                     // Table 2:86
1178        struct {
1179            UINT16              size;
1180            BYTE                buffer[1024];
1181        }           t;
```

```
1182        TPM2B        b;
1183    } TPM2B_EVENT;                                  /* Structure */
1184    typedef union {                                 // Table 2:87
1185        struct {
1186            UINT16              size;
1187            BYTE                buffer[MAX_DIGEST_BUFFER];
1188        }            t;
1189        TPM2B        b;
1190    } TPM2B_MAX_BUFFER;                             /* Structure */
1191    typedef union {                                 // Table 2:88
1192        struct {
1193            UINT16              size;
1194            BYTE                buffer[MAX_NV_BUFFER_SIZE];
1195        }            t;
1196        TPM2B        b;
1197    } TPM2B_MAX_NV_BUFFER;                          /* Structure */
1198    typedef union {                                 // Table 2:89
1199        struct {
1200            UINT16              size;
1201            BYTE                buffer[sizeof(UINT64)];
1202        }            t;
1203        TPM2B        b;
1204    } TPM2B_TIMEOUT;                                /* Structure */
1205    typedef union {                                 // Table 2:90
1206        struct {
1207            UINT16              size;
1208            BYTE                buffer[MAX_SYM_BLOCK_SIZE];
1209        }            t;
1210        TPM2B        b;
1211    } TPM2B_IV;                                     /* Structure */
1212    typedef union {                                 // Table 2:91
1213        TPMT_HA                 digest;
1214        TPM_HANDLE              handle;
1215    } TPMU_NAME;                                    /* Structure */
1216    typedef union {                                 // Table 2:92
1217        struct {
1218            UINT16              size;
1219            BYTE                name[sizeof(TPMU_NAME)];
1220        }            t;
1221        TPM2B        b;
1222    } TPM2B_NAME;                                   /* Structure */
1223    typedef struct {                                // Table 2:93
1224        UINT8                   sizeofSelect;
1225        BYTE                    pcrSelect[PCR_SELECT_MAX];
1226    } TPMS_PCR_SELECT;                              /* Structure */
1227    typedef struct {                                // Table 2:94
1228        TPMI_ALG_HASH           hash;
1229        UINT8                   sizeofSelect;
1230        BYTE                    pcrSelect[PCR_SELECT_MAX];
1231    } TPMS_PCR_SELECTION;                           /* Structure */
1232    typedef struct {                                // Table 2:97
1233        TPM_ST                  tag;
1234        TPMI_RH_HIERARCHY       hierarchy;
1235        TPM2B_DIGEST            digest;
1236    } TPMT_TK_CREATION;                             /* Structure */
1237    typedef struct {                                // Table 2:98
1238        TPM_ST                  tag;
1239        TPMI_RH_HIERARCHY       hierarchy;
1240        TPM2B_DIGEST            digest;
1241    } TPMT_TK_VERIFIED;                             /* Structure */
1242    typedef struct {                                // Table 2:99
1243        TPM_ST                  tag;
1244        TPMI_RH_HIERARCHY       hierarchy;
1245        TPM2B_DIGEST            digest;
1246    } TPMT_TK_AUTH;                                 /* Structure */
1247    typedef struct {                                // Table 2:100
```

```
1248        TPM_ST                  tag;
1249        TPMI_RH_HIERARCHY       hierarchy;
1250        TPM2B_DIGEST            digest;
1251    } TPMT_TK_HASHCHECK;                            /* Structure */
1252    typedef struct {                                // Table 2:101
1253        TPM_ALG_ID              alg;
1254        TPMA_ALGORITHM          algProperties;
1255    } TPMS_ALG_PROPERTY;                            /* Structure */
1256    typedef struct {                                // Table 2:102
1257        TPM_PT                  property;
1258        UINT32                  value;
1259    } TPMS_TAGGED_PROPERTY;                         /* Structure */
1260    typedef struct {                                // Table 2:103
1261        TPM_PT_PCR              tag;
1262        UINT8                   sizeofSelect;
1263        BYTE                    pcrSelect[PCR_SELECT_MAX];
1264    } TPMS_TAGGED_PCR_SELECT;                       /* Structure */
1265    typedef struct {                                // Table 2:104
1266        TPM_HANDLE              handle;
1267        TPMT_HA                 policyHash;
1268    } TPMS_TAGGED_POLICY;                           /* Structure */
1269    typedef struct {                                // Table 2:105
1270        TPM_HANDLE              handle;
1271        UINT32                  timeout;
1272        TPMA_ACT                attributes;
1273    } TPMS_ACT_DATA;                                /* Structure */
1274    typedef struct {                                // Table 2:106
1275        UINT32                  count;
1276        TPM_CC                  commandCodes[MAX_CAP_CC];
1277    } TPML_CC;                                      /* Structure */
1278    typedef struct {                                // Table 2:107
1279        UINT32                  count;
1280        TPMA_CC                 commandAttributes[MAX_CAP_CC];
1281    } TPML_CCA;                                     /* Structure */
1282    typedef struct {                                // Table 2:108
1283        UINT32                  count;
1284        TPM_ALG_ID              algorithms[MAX_ALG_LIST_SIZE];
1285    } TPML_ALG;                                     /* Structure */
1286    typedef struct {                                // Table 2:109
1287        UINT32                  count;
1288        TPM_HANDLE              handle[MAX_CAP_HANDLES];
1289    } TPML_HANDLE;                                  /* Structure */
1290    typedef struct {                                // Table 2:110
1291        UINT32                  count;
1292        TPM2B_DIGEST            digests[8];
1293    } TPML_DIGEST;                                  /* Structure */
1294    typedef struct {                                // Table 2:111
1295        UINT32                  count;
1296        TPMT_HA                 digests[HASH_COUNT];
1297    } TPML_DIGEST_VALUES;                           /* Structure */
1298    typedef struct {                                // Table 2:112
1299        UINT32                  count;
1300        TPMS_PCR_SELECTION      pcrSelections[HASH_COUNT];
1301    } TPML_PCR_SELECTION;                           /* Structure */
1302    typedef struct {                                // Table 2:113
1303        UINT32                  count;
1304        TPMS_ALG_PROPERTY       algProperties[MAX_CAP_ALGS];
1305    } TPML_ALG_PROPERTY;                            /* Structure */
1306    typedef struct {                                // Table 2:114
1307        UINT32                      count;
1308        TPMS_TAGGED_PROPERTY        tpmProperty[MAX_TPM_PROPERTIES];
1309    } TPML_TAGGED_TPM_PROPERTY;                     /* Structure */
1310    typedef struct {                                // Table 2:115
1311        UINT32                      count;
1312        TPMS_TAGGED_PCR_SELECT      pcrProperty[MAX_PCR_PROPERTIES];
1313    } TPML_TAGGED_PCR_PROPERTY;                     /* Structure */
```

```
1314    typedef struct {                                    // Table 2:116
1315        UINT32                  count;
1316        TPM_ECC_CURVE           eccCurves[MAX_ECC_CURVES];
1317    } TPML_ECC_CURVE;                                   /* Structure */
1318    typedef struct {                                    // Table 2:117
1319        UINT32                  count;
1320        TPMS_TAGGED_POLICY      policies[MAX_TAGGED_POLICIES];
1321    } TPML_TAGGED_POLICY;                               /* Structure */
1322    typedef struct {                                    // Table 2:118
1323        UINT32                  count;
1324        TPMS_ACT_DATA           actData[MAX_ACT_DATA];
1325    } TPML_ACT_DATA;                                    /* Structure */
1326    typedef union {                                     // Table 2:119
1327        TPML_ALG_PROPERTY               algorithms;
1328        TPML_HANDLE                     handles;
1329        TPML_CCA                        command;
1330        TPML_CC                         ppCommands;
1331        TPML_CC                         auditCommands;
1332        TPML_PCR_SELECTION              assignedPCR;
1333        TPML_TAGGED_TPM_PROPERTY        tpmProperties;
1334        TPML_TAGGED_PCR_PROPERTY        pcrProperties;
1335    #if ALG_ECC
1336        TPML_ECC_CURVE                  eccCurves;
1337    #endif // ALG_ECC
1338        TPML_TAGGED_POLICY              authPolicies;
1339        TPML_ACT_DATA                   actData;
1340    } TPMU_CAPABILITIES;                                /* Structure */
1341    typedef struct {                                    // Table 2:120
1342        TPM_CAP                 capability;
1343        TPMU_CAPABILITIES       data;
1344    } TPMS_CAPABILITY_DATA;                             /* Structure */
1345    typedef struct {                                    // Table 2:121
1346        UINT64                  clock;
1347        UINT32                  resetCount;
1348        UINT32                  restartCount;
1349        TPMI_YES_NO             safe;
1350    } TPMS_CLOCK_INFO;                                  /* Structure */
1351    typedef struct {                                    // Table 2:122
1352        UINT64                  time;
1353        TPMS_CLOCK_INFO         clockInfo;
1354    } TPMS_TIME_INFO;                                   /* Structure */
1355    typedef struct {                                    // Table 2:123
1356        TPMS_TIME_INFO          time;
1357        UINT64                  firmwareVersion;
1358    } TPMS_TIME_ATTEST_INFO;                            /* Structure */
1359    typedef struct {                                    // Table 2:124
1360        TPM2B_NAME              name;
1361        TPM2B_NAME              qualifiedName;
1362    } TPMS_CERTIFY_INFO;                                /* Structure */
1363    typedef struct {                                    // Table 2:125
1364        TPML_PCR_SELECTION      pcrSelect;
1365        TPM2B_DIGEST            pcrDigest;
1366    } TPMS_QUOTE_INFO;                                  /* Structure */
1367    typedef struct {                                    // Table 2:126
1368        UINT64                  auditCounter;
1369        TPM_ALG_ID              digestAlg;
1370        TPM2B_DIGEST            auditDigest;
1371        TPM2B_DIGEST            commandDigest;
1372    } TPMS_COMMAND_AUDIT_INFO;                          /* Structure */
1373    typedef struct {                                    // Table 2:127
1374        TPMI_YES_NO             exclusiveSession;
1375        TPM2B_DIGEST            sessionDigest;
1376    } TPMS_SESSION_AUDIT_INFO;                          /* Structure */
1377    typedef struct {                                    // Table 2:128
1378        TPM2B_NAME              objectName;
1379        TPM2B_DIGEST            creationHash;
```

```
1380    } TPMS_CREATION_INFO;                                   /* Structure */
1381    typedef struct {                                        // Table 2:129
1382        TPM2B_NAME                  indexName;
1383        UINT16                      offset;
1384        TPM2B_MAX_NV_BUFFER         nvContents;
1385    } TPMS_NV_CERTIFY_INFO;                                 /* Structure */
1386    typedef struct {                                        // Table 2:130
1387        TPM2B_NAME              indexName;
1388        TPM2B_DIGEST            nvDigest;
1389    } TPMS_NV_DIGEST_CERTIFY_INFO;                          /* Structure */
1390    typedef TPM_ST              TPMI_ST_ATTEST;             // Table 2:131  /* Interface */
1391    typedef union {                                                 // Table 2:132
1392        TPMS_CERTIFY_INFO                   certify;
1393        TPMS_CREATION_INFO                  creation;
1394        TPMS_QUOTE_INFO                     quote;
1395        TPMS_COMMAND_AUDIT_INFO             commandAudit;
1396        TPMS_SESSION_AUDIT_INFO             sessionAudit;
1397        TPMS_TIME_ATTEST_INFO               time;
1398        TPMS_NV_CERTIFY_INFO                nv;
1399        TPMS_NV_DIGEST_CERTIFY_INFO         nvDigest;
1400    } TPMU_ATTEST;                                                  /* Structure */
1401    typedef struct {                                        // Table 2:133
1402        TPM_GENERATED           magic;
1403        TPMI_ST_ATTEST          type;
1404        TPM2B_NAME              qualifiedSigner;
1405        TPM2B_DATA              extraData;
1406        TPMS_CLOCK_INFO         clockInfo;
1407        UINT64                  firmwareVersion;
1408        TPMU_ATTEST             attested;
1409    } TPMS_ATTEST;                                          /* Structure */
1410    typedef union {                                         // Table 2:134
1411        struct {
1412            UINT16              size;
1413            BYTE                attestationData[sizeof(TPMS_ATTEST)];
1414        }               t;
1415        TPM2B           b;
1416    } TPM2B_ATTEST;                                         /* Structure */
1417    typedef struct {                                        // Table 2:135
1418        TPMI_SH_AUTH_SESSION        sessionHandle;
1419        TPM2B_NONCE                 nonce;
1420        TPMA_SESSION                sessionAttributes;
1421        TPM2B_AUTH                  hmac;
1422    } TPMS_AUTH_COMMAND;                                    /* Structure */
1423    typedef struct {                                        // Table 2:136
1424        TPM2B_NONCE             nonce;
1425        TPMA_SESSION            sessionAttributes;
1426        TPM2B_AUTH              hmac;
1427    } TPMS_AUTH_RESPONSE;                                   /* Structure */
1428    typedef TPM_KEY_BITS        TPMI_TDES_KEY_BITS;     // Table 2:137  /* Interface */
1429    typedef TPM_KEY_BITS        TPMI_AES_KEY_BITS;      // Table 2:137  /* Interface */
1430    typedef TPM_KEY_BITS        TPMI_SM4_KEY_BITS;      // Table 2:137  /* Interface */
1431    typedef TPM_KEY_BITS        TPMI_CAMELLIA_KEY_BITS; // Table 2:137  /* Interface */
1432    typedef union {                                         // Table 2:138
1433    #if ALG_TDES
1434        TPMI_TDES_KEY_BITS          tdes;
1435    #endif // ALG_TDES
1436    #if ALG_AES
1437        TPMI_AES_KEY_BITS           aes;
1438    #endif // ALG_AES
1439    #if ALG_SM4
1440        TPMI_SM4_KEY_BITS           sm4;
1441    #endif // ALG_SM4
1442    #if ALG_CAMELLIA
1443        TPMI_CAMELLIA_KEY_BITS      camellia;
1444    #endif // ALG_CAMELLIA
1445        TPM_KEY_BITS                sym;
```

```
1446    #if ALG_XOR
1447        TPMI_ALG_HASH               xor;
1448    #endif // ALG_XOR
1449    } TPMU_SYM_KEY_BITS;                                    /* Structure */
1450    typedef union {                                         // Table 2:139
1451    #if ALG_TDES
1452        TPMI_ALG_SYM_MODE       tdes;
1453    #endif // ALG_TDES
1454    #if ALG_AES
1455        TPMI_ALG_SYM_MODE       aes;
1456    #endif // ALG_AES
1457    #if ALG_SM4
1458        TPMI_ALG_SYM_MODE       sm4;
1459    #endif // ALG_SM4
1460    #if ALG_CAMELLIA
1461        TPMI_ALG_SYM_MODE       camellia;
1462    #endif // ALG_CAMELLIA
1463        TPMI_ALG_SYM_MODE       sym;
1464    } TPMU_SYM_MODE;                                        /* Structure */
1465    typedef struct {                                        // Table 2:141
1466        TPMI_ALG_SYM            algorithm;
1467        TPMU_SYM_KEY_BITS       keyBits;
1468        TPMU_SYM_MODE           mode;
1469    } TPMT_SYM_DEF;                                         /* Structure */
1470    typedef struct {                                        // Table 2:142
1471        TPMI_ALG_SYM_OBJECT         algorithm;
1472        TPMU_SYM_KEY_BITS           keyBits;
1473        TPMU_SYM_MODE               mode;
1474    } TPMT_SYM_DEF_OBJECT;                                  /* Structure */
1475    typedef union {                                         // Table 2:143
1476        struct {
1477            UINT16              size;
1478            BYTE                buffer[MAX_SYM_KEY_BYTES];
1479        }               t;
1480        TPM2B           b;
1481    } TPM2B_SYM_KEY;                                        /* Structure */
1482    typedef struct {                                        // Table 2:144
1483        TPMT_SYM_DEF_OBJECT         sym;
1484    } TPMS_SYMCIPHER_PARMS;                                 /* Structure */
1485    typedef union {                                         // Table 2:145
1486        struct {
1487            UINT16              size;
1488            BYTE                buffer[LABEL_MAX_BUFFER];
1489        }               t;
1490        TPM2B           b;
1491    } TPM2B_LABEL;                                          /* Structure */
1492    typedef struct {                                        // Table 2:146
1493        TPM2B_LABEL             label;
1494        TPM2B_LABEL             context;
1495    } TPMS_DERIVE;                                          /* Structure */
1496    typedef union {                                         // Table 2:147
1497        struct {
1498            UINT16              size;
1499            BYTE                buffer[sizeof(TPMS_DERIVE)];
1500        }               t;
1501        TPM2B           b;
1502    } TPM2B_DERIVE;                                         /* Structure */
1503    typedef union {                                         // Table 2:148
1504        BYTE                    create[MAX_SYM_DATA];
1505        TPMS_DERIVE             derive;
1506    } TPMU_SENSITIVE_CREATE;                                /* Structure */
1507    typedef union {                                         // Table 2:149
1508        struct {
1509            UINT16              size;
1510            BYTE                buffer[sizeof(TPMU_SENSITIVE_CREATE)];
1511        }               t;
```

```
1512        TPM2B        b;
1513    } TPM2B_SENSITIVE_DATA;                                 /* Structure */
1514    typedef struct {                                        // Table 2:150
1515        TPM2B_AUTH                  userAuth;
1516        TPM2B_SENSITIVE_DATA        data;
1517    } TPMS_SENSITIVE_CREATE;                                /* Structure */
1518    typedef struct {                                        // Table 2:151
1519        UINT16                      size;
1520        TPMS_SENSITIVE_CREATE       sensitive;
1521    } TPM2B_SENSITIVE_CREATE;                               /* Structure */
1522    typedef struct {                                        // Table 2:152
1523        TPMI_ALG_HASH           hashAlg;
1524    } TPMS_SCHEME_HASH;                                     /* Structure */
1525    typedef struct {                                        // Table 2:153
1526        TPMI_ALG_HASH           hashAlg;
1527        UINT16                  count;
1528    } TPMS_SCHEME_ECDAA;                                    /* Structure */
1529    typedef TPM_ALG_ID          TPMI_ALG_KEYEDHASH_SCHEME;
```

Table 2:155 - Definition of Types for HMAC_SIG_SCHEME

```
1530    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_HMAC;
1531    typedef struct {                                        // Table 2:156
1532        TPMI_ALG_HASH           hashAlg;
1533        TPMI_ALG_KDF            kdf;
1534    } TPMS_SCHEME_XOR;                                      /* Structure */
1535    typedef union {                                         // Table 2:157
1536    #if ALG_HMAC
1537        TPMS_SCHEME_HMAC        hmac;
1538    #endif // ALG_HMAC
1539    #if ALG_XOR
1540        TPMS_SCHEME_XOR         xor;
1541    #endif // ALG_XOR
1542    } TPMU_SCHEME_KEYEDHASH;                                /* Structure */
1543    typedef struct {                                        // Table 2:158
1544        TPMI_ALG_KEYEDHASH_SCHEME       scheme;
1545        TPMU_SCHEME_KEYEDHASH           details;
1546    } TPMT_KEYEDHASH_SCHEME;                                /* Structure */
```

Table 2:159 - Definition of Types for RSA Signature Schemes

```
1547    typedef TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_RSASSA;
1548    typedef TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_RSAPSS;
```

Table 2:160 - Definition of Types for ECC Signature Schemes

```
1549    typedef TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_ECDSA;
1550    typedef TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_SM2;
1551    typedef TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_ECSCHNORR;
1552    typedef TPMS_SCHEME_ECDAA   TPMS_SIG_SCHEME_ECDAA;
1553    typedef union {                                         // Table 2:161
1554    #if ALG_ECC
1555        TPMS_SIG_SCHEME_ECDAA           ecdaa;
1556    #endif // ALG_ECC
1557    #if ALG_RSASSA
1558        TPMS_SIG_SCHEME_RSASSA          rsassa;
1559    #endif // ALG_RSASSA
1560    #if ALG_RSAPSS
1561        TPMS_SIG_SCHEME_RSAPSS          rsapss;
1562    #endif // ALG_RSAPSS
1563    #if ALG_ECDSA
1564        TPMS_SIG_SCHEME_ECDSA           ecdsa;
1565    #endif // ALG_ECDSA
1566    #if ALG_SM2
```

```
1567        TPMS_SIG_SCHEME_SM2             sm2;
1568    #endif // ALG_SM2
1569    #if ALG_ECSCHNORR
1570        TPMS_SIG_SCHEME_ECSCHNORR       ecschnorr;
1571    #endif // ALG_ECSCHNORR
1572    #if ALG_HMAC
1573        TPMS_SCHEME_HMAC                hmac;
1574    #endif // ALG_HMAC
1575        TPMS_SCHEME_HASH                any;
1576    } TPMU_SIG_SCHEME;                                  /* Structure */
1577    typedef struct {                                    // Table 2:162
1578        TPMI_ALG_SIG_SCHEME         scheme;
1579        TPMU_SIG_SCHEME             details;
1580    } TPMT_SIG_SCHEME;                                  /* Structure */
```

Table 2:163 - Definition of Types for Encryption Schemes

```
1581    typedef TPMS_SCHEME_HASH    TPMS_ENC_SCHEME_OAEP;
1582    typedef TPMS_EMPTY          TPMS_ENC_SCHEME_RSAES;
```

Table 2:164 - Definition of Types for ECC Key Exchange

```
1583    typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECDH;
1584    typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECMQV;
```

Table 2:165 - Definition of Types for KDF Schemes

```
1585    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_MGF1;
1586    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_56A;
1587    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF2;
1588    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_108;
1589    typedef union {                                     // Table 2:166
1590    #if ALG_MGF1
1591        TPMS_SCHEME_MGF1                mgf1;
1592    #endif // ALG_MGF1
1593    #if ALG_KDF1_SP800_56A
1594        TPMS_SCHEME_KDF1_SP800_56A      kdf1_sp800_56a;
1595    #endif // ALG_KDF1_SP800_56A
1596    #if ALG_KDF2
1597        TPMS_SCHEME_KDF2                kdf2;
1598    #endif // ALG_KDF2
1599    #if ALG_KDF1_SP800_108
1600        TPMS_SCHEME_KDF1_SP800_108      kdf1_sp800_108;
1601    #endif // ALG_KDF1_SP800_108
1602    } TPMU_KDF_SCHEME;                                  /* Structure */
1603    typedef struct {                                    // Table 2:167
1604        TPMI_ALG_KDF            scheme;
1605        TPMU_KDF_SCHEME         details;
1606    } TPMT_KDF_SCHEME;                                  /* Structure */
1607    typedef TPM_ALG_ID          TPMI_ALG_ASYM_SCHEME;   // Table 2:168  /* Interface */
1608    typedef union {                                     // Table 2:169
1609    #if ALG_ECDH
1610        TPMS_KEY_SCHEME_ECDH            ecdh;
1611    #endif // ALG_ECDH
1612    #if ALG_ECMQV
1613        TPMS_KEY_SCHEME_ECMQV           ecmqv;
1614    #endif // ALG_ECMQV
1615    #if ALG_ECC
1616        TPMS_SIG_SCHEME_ECDAA           ecdaa;
1617    #endif // ALG_ECC
1618    #if ALG_RSASSA
1619        TPMS_SIG_SCHEME_RSASSA          rsassa;
1620    #endif // ALG_RSASSA
1621    #if ALG_RSAPSS
```

```
1622        TPMS_SIG_SCHEME_RSAPSS          rsapss;
1623    #endif // ALG_RSAPSS
1624    #if ALG_ECDSA
1625        TPMS_SIG_SCHEME_ECDSA           ecdsa;
1626    #endif // ALG_ECDSA
1627    #if ALG_SM2
1628        TPMS_SIG_SCHEME_SM2             sm2;
1629    #endif // ALG_SM2
1630    #if ALG_ECSCHNORR
1631        TPMS_SIG_SCHEME_ECSCHNORR       ecschnorr;
1632    #endif // ALG_ECSCHNORR
1633    #if ALG_RSAES
1634        TPMS_ENC_SCHEME_RSAES           rsaes;
1635    #endif // ALG_RSAES
1636    #if ALG_OAEP
1637        TPMS_ENC_SCHEME_OAEP            oaep;
1638    #endif // ALG_OAEP
1639        TPMS_SCHEME_HASH                anySig;
1640    } TPMU_ASYM_SCHEME;                                 /* Structure */
1641    typedef struct {                                    // Table 2:170
1642        TPMI_ALG_ASYM_SCHEME        scheme;
1643        TPMU_ASYM_SCHEME            details;
1644    } TPMT_ASYM_SCHEME;                                 /* Structure */
1645    typedef TPM_ALG_ID          TPMI_ALG_RSA_SCHEME;    // Table 2:171  /* Interface */
1646    typedef struct {                                    // Table 2:172
1647        TPMI_ALG_RSA_SCHEME         scheme;
1648        TPMU_ASYM_SCHEME            details;
1649    } TPMT_RSA_SCHEME;                                  /* Structure */
1650    typedef TPM_ALG_ID          TPMI_ALG_RSA_DECRYPT;   // Table 2:173  /* Interface */
1651    typedef struct {                                    // Table 2:174
1652        TPMI_ALG_RSA_DECRYPT        scheme;
1653        TPMU_ASYM_SCHEME            details;
1654    } TPMT_RSA_DECRYPT;                                 /* Structure */
1655    typedef union {                                     // Table 2:175
1656        struct {
1657            UINT16              size;
1658            BYTE                buffer[MAX_RSA_KEY_BYTES];
1659        }               t;
1660        TPM2B           b;
1661    } TPM2B_PUBLIC_KEY_RSA;                             /* Structure */
1662    typedef TPM_KEY_BITS        TPMI_RSA_KEY_BITS;      // Table 2:176  /* Interface */
1663    typedef union {                                     // Table 2:177
1664        struct {
1665            UINT16              size;
1666            BYTE                buffer[RSA_PRIVATE_SIZE];
1667        }               t;
1668        TPM2B           b;
1669    } TPM2B_PRIVATE_KEY_RSA;                            /* Structure */
1670    typedef union {                                     // Table 2:178
1671        struct {
1672            UINT16              size;
1673            BYTE                buffer[MAX_ECC_KEY_BYTES];
1674        }               t;
1675        TPM2B           b;
1676    } TPM2B_ECC_PARAMETER;                              /* Structure */
1677    typedef struct {                                    // Table 2:179
1678        TPM2B_ECC_PARAMETER         x;
1679        TPM2B_ECC_PARAMETER         y;
1680    } TPMS_ECC_POINT;                                   /* Structure */
1681    typedef struct {                                    // Table 2:180
1682        UINT16                  size;
1683        TPMS_ECC_POINT          point;
1684    } TPM2B_ECC_POINT;                                  /* Structure */
1685    typedef TPM_ALG_ID          TPMI_ALG_ECC_SCHEME;    // Table 2:181  /* Interface */
1686    typedef TPM_ECC_CURVE       TPMI_ECC_CURVE;         // Table 2:182  /* Interface */
1687    typedef struct {                                    // Table 2:183
```

```
1688        TPMI_ALG_ECC_SCHEME         scheme;
1689        TPMU_ASYM_SCHEME            details;
1690    } TPMT_ECC_SCHEME;                                  /* Structure */
1691    typedef struct {                                    // Table 2:184
1692        TPM_ECC_CURVE               curveID;
1693        UINT16                      keySize;
1694        TPMT_KDF_SCHEME             kdf;
1695        TPMT_ECC_SCHEME             sign;
1696        TPM2B_ECC_PARAMETER         p;
1697        TPM2B_ECC_PARAMETER         a;
1698        TPM2B_ECC_PARAMETER         b;
1699        TPM2B_ECC_PARAMETER         gX;
1700        TPM2B_ECC_PARAMETER         gY;
1701        TPM2B_ECC_PARAMETER         n;
1702        TPM2B_ECC_PARAMETER         h;
1703    } TPMS_ALGORITHM_DETAIL_ECC;                        /* Structure */
1704    typedef struct {                                    // Table 2:185
1705        TPMI_ALG_HASH               hash;
1706        TPM2B_PUBLIC_KEY_RSA        sig;
1707    } TPMS_SIGNATURE_RSA;                               /* Structure */
```

Table 2:186 - Definition of Types for Signature

```
1708    typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSASSA;
1709    typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSAPSS;
1710    typedef struct {                                    // Table 2:187
1711        TPMI_ALG_HASH               hash;
1712        TPM2B_ECC_PARAMETER         signatureR;
1713        TPM2B_ECC_PARAMETER         signatureS;
1714    } TPMS_SIGNATURE_ECC;                               /* Structure */
```

Table 2:188 - Definition of Types for TPMS_SIGNATURE_ECC

```
1715    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECDAA;
1716    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECDSA;
1717    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_SM2;
1718    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECSCHNORR;
1719    typedef union {                                     // Table 2:189
1720    #if ALG_ECC
1721        TPMS_SIGNATURE_ECDAA            ecdaa;
1722    #endif // ALG_ECC
1723    #if ALG_RSA
1724        TPMS_SIGNATURE_RSASSA           rsassa;
1725    #endif // ALG_RSA
1726    #if ALG_RSA
1727        TPMS_SIGNATURE_RSAPSS           rsapss;
1728    #endif // ALG_RSA
1729    #if ALG_ECC
1730        TPMS_SIGNATURE_ECDSA            ecdsa;
1731    #endif // ALG_ECC
1732    #if ALG_ECC
1733        TPMS_SIGNATURE_SM2              sm2;
1734    #endif // ALG_ECC
1735    #if ALG_ECC
1736        TPMS_SIGNATURE_ECSCHNORR        ecschnorr;
1737    #endif // ALG_ECC
1738    #if ALG_HMAC
1739        TPMT_HA                         hmac;
1740    #endif // ALG_HMAC
1741        TPMS_SCHEME_HASH                any;
1742    } TPMU_SIGNATURE;                                   /* Structure */
1743    typedef struct {                                    // Table 2:190
1744        TPMI_ALG_SIG_SCHEME         sigAlg;
1745        TPMU_SIGNATURE              signature;
1746    } TPMT_SIGNATURE;                                   /* Structure */
```

```
1747    typedef union {                                 // Table 2:191
1748    #if ALG_ECC
1749        BYTE                    ecc[sizeof(TPMS_ECC_POINT)];
1750    #endif // ALG_ECC
1751    #if ALG_RSA
1752        BYTE                    rsa[MAX_RSA_KEY_BYTES];
1753    #endif // ALG_RSA
1754    #if ALG_SYMCIPHER
1755        BYTE                    symmetric[sizeof(TPM2B_DIGEST)];
1756    #endif // ALG_SYMCIPHER
1757    #if ALG_KEYEDHASH
1758        BYTE                    keyedHash[sizeof(TPM2B_DIGEST)];
1759    #endif // ALG_KEYEDHASH
1760    } TPMU_ENCRYPTED_SECRET;                        /* Structure */
1761    typedef union {                                 // Table 2:192
1762        struct {
1763            UINT16              size;
1764            BYTE                secret[sizeof(TPMU_ENCRYPTED_SECRET)];
1765        }               t;
1766        TPM2B           b;
1767    } TPM2B_ENCRYPTED_SECRET;                       /* Structure */
1768    typedef TPM_ALG_ID          TPMI_ALG_PUBLIC;    // Table 2:193  /* Interface */
1769    typedef union {                                 // Table 2:194
1770    #if ALG_KEYEDHASH
1771        TPM2B_DIGEST                keyedHash;
1772    #endif // ALG_KEYEDHASH
1773    #if ALG_SYMCIPHER
1774        TPM2B_DIGEST                sym;
1775    #endif // ALG_SYMCIPHER
1776    #if ALG_RSA
1777        TPM2B_PUBLIC_KEY_RSA        rsa;
1778    #endif // ALG_RSA
1779    #if ALG_ECC
1780        TPMS_ECC_POINT              ecc;
1781    #endif // ALG_ECC
1782        TPMS_DERIVE                 derive;
1783    } TPMU_PUBLIC_ID;                               /* Structure */
1784    typedef struct {                                // Table 2:195
1785        TPMT_KEYEDHASH_SCHEME       scheme;
1786    } TPMS_KEYEDHASH_PARMS;                         /* Structure */
1787    typedef struct {                                // Table 2:196
1788        TPMT_SYM_DEF_OBJECT         symmetric;
1789        TPMT_ASYM_SCHEME            scheme;
1790    } TPMS_ASYM_PARMS;                              /* Structure */
1791    typedef struct {                                // Table 2:197
1792        TPMT_SYM_DEF_OBJECT         symmetric;
1793        TPMT_RSA_SCHEME             scheme;
1794        TPMI_RSA_KEY_BITS           keyBits;
1795        UINT32                      exponent;
1796    } TPMS_RSA_PARMS;                               /* Structure */
1797    typedef struct {                                // Table 2:198
1798        TPMT_SYM_DEF_OBJECT         symmetric;
1799        TPMT_ECC_SCHEME             scheme;
1800        TPMI_ECC_CURVE              curveID;
1801        TPMT_KDF_SCHEME             kdf;
1802    } TPMS_ECC_PARMS;                               /* Structure */
1803    typedef union {                                 // Table 2:199
1804    #if ALG_KEYEDHASH
1805        TPMS_KEYEDHASH_PARMS        keyedHashDetail;
1806    #endif // ALG_KEYEDHASH
1807    #if ALG_SYMCIPHER
1808        TPMS_SYMCIPHER_PARMS        symDetail;
1809    #endif // ALG_SYMCIPHER
1810    #if ALG_RSA
1811        TPMS_RSA_PARMS              rsaDetail;
1812    #endif // ALG_RSA
```



```
1813    #if ALG_ECC
1814        TPMS_ECC_PARMS              eccDetail;
1815    #endif // ALG_ECC
1816        TPMS_ASYM_PARMS             asymDetail;
1817    } TPMU_PUBLIC_PARMS;                                    /* Structure */
1818    typedef struct {                                        // Table 2:200
1819        TPMI_ALG_PUBLIC         type;
1820        TPMU_PUBLIC_PARMS       parameters;
1821    } TPMT_PUBLIC_PARMS;                                    /* Structure */
1822    typedef struct {                                        // Table 2:201
1823        TPMI_ALG_PUBLIC         type;
1824        TPMI_ALG_HASH           nameAlg;
1825        TPMA_OBJECT             objectAttributes;
1826        TPM2B_DIGEST            authPolicy;
1827        TPMU_PUBLIC_PARMS       parameters;
1828        TPMU_PUBLIC_ID          unique;
1829    } TPMT_PUBLIC;                                          /* Structure */
1830    typedef struct {                                        // Table 2:202
1831        UINT16                  size;
1832        TPMT_PUBLIC             publicArea;
1833    } TPM2B_PUBLIC;                                         /* Structure */
1834    typedef union {                                         // Table 2:203
1835        struct {
1836            UINT16              size;
1837            BYTE                buffer[sizeof(TPMT_PUBLIC)];
1838        }           t;
1839        TPM2B       b;
1840    } TPM2B_TEMPLATE;                                       /* Structure */
1841    typedef union {                                         // Table 2:204
1842        struct {
1843            UINT16              size;
1844            BYTE                buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
1845        }           t;
1846        TPM2B       b;
1847    } TPM2B_PRIVATE_VENDOR_SPECIFIC;                        /* Structure */
1848    typedef union {                                         // Table 2:205
1849    #if ALG_RSA
1850        TPM2B_PRIVATE_KEY_RSA               rsa;
1851    #endif // ALG_RSA
1852    #if ALG_ECC
1853        TPM2B_ECC_PARAMETER                 ecc;
1854    #endif // ALG_ECC
1855    #if ALG_KEYEDHASH
1856        TPM2B_SENSITIVE_DATA                bits;
1857    #endif // ALG_KEYEDHASH
1858    #if ALG_SYMCIPHER
1859        TPM2B_SYM_KEY                       sym;
1860    #endif // ALG_SYMCIPHER
1861        TPM2B_PRIVATE_VENDOR_SPECIFIC       any;
1862    } TPMU_SENSITIVE_COMPOSITE;                             /* Structure */
1863    typedef struct {                                        // Table 2:206
1864        TPMI_ALG_PUBLIC                 sensitiveType;
1865        TPM2B_AUTH                      authValue;
1866        TPM2B_DIGEST                    seedValue;
1867        TPMU_SENSITIVE_COMPOSITE        sensitive;
1868    } TPMT_SENSITIVE;                                       /* Structure */
1869    typedef struct {                                        // Table 2:207
1870        UINT16                  size;
1871        TPMT_SENSITIVE          sensitiveArea;
1872    } TPM2B_SENSITIVE;                                      /* Structure */
1873    typedef struct {                                        // Table 2:208
1874        TPM2B_DIGEST            integrityOuter;
1875        TPM2B_DIGEST            integrityInner;
1876        TPM2B_SENSITIVE         sensitive;
1877    } _PRIVATE;                                             /* Structure */
1878    typedef union {                                         // Table 2:209
```

```
1879        struct {
1880            UINT16              size;
1881            BYTE                buffer[sizeof(_PRIVATE)];
1882        }           t;
1883        TPM2B       b;
1884    } TPM2B_PRIVATE;                                    /* Structure */
1885    typedef struct {                                    // Table 2:210
1886        TPM2B_DIGEST            integrityHMAC;
1887        TPM2B_DIGEST            encIdentity;
1888    } TPMS_ID_OBJECT;                                   /* Structure */
1889    typedef union {                                     // Table 2:211
1890        struct {
1891            UINT16              size;
1892            BYTE                credential[sizeof(TPMS_ID_OBJECT)];
1893        }           t;
1894        TPM2B       b;
1895    } TPM2B_ID_OBJECT;                                  /* Structure */
1896    #define TYPE_OF_TPM_NV_INDEX    UINT32
1897    #define TPM_NV_INDEX_TO_UINT32(a)     (*((UINT32 *)&(a)))
1898    #define UINT32_TO_TPM_NV_INDEX(a)     (*((TPM_NV_INDEX *)&(a)))
1899    #define TPM_NV_INDEX_TO_BYTE_ARRAY(i, a)                                    \
1900                UINT32_TO_BYTE_ARRAY((TPM_NV_INDEX_TO_UINT32(i)), (a))
1901    #define BYTE_ARRAY_TO_TPM_NV_INDEX(i, a)                                    \
1902                { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPM_NV_INDEX(x); }
1903    #if USE_BIT_FIELD_STRUCTURES
1904    typedef struct TPM_NV_INDEX {                       // Table 2:212
1905        unsigned    index               : 24;
1906        unsigned    RH_NV               : 8;
1907    } TPM_NV_INDEX;                                     /* Bits */
```

This is the initializer for a TPM_NV_INDEX structure

```
1908    #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {index, rh_nv}
1909    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:212 TPM_NV_INDEX using bit masking

```
1910    typedef UINT32                      TPM_NV_INDEX;
1911    #define TYPE_OF_TPM_NV_INDEX        UINT32
1912    #define TPM_NV_INDEX_index_SHIFT    0
1913    #define TPM_NV_INDEX_index          ((TPM_NV_INDEX)0xffffff << 0)
1914    #define TPM_NV_INDEX_RH_NV_SHIFT    24
1915    #define TPM_NV_INDEX_RH_NV          ((TPM_NV_INDEX)0xff << 24)
```

This is the initializer for a TPM_NV_INDEX bit array.

```
1916    #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {(index << 0) + (rh_nv << 24)}
1917    #endif // USE_BIT_FIELD_STRUCTURES
```

Table 2:213 - Definition of TPM_NT Constants

```
1918    typedef UINT32              TPM_NT;
1919    #define TYPE_OF_TPM_NT      UINT32
1920    #define TPM_NT_ORDINARY     (TPM_NT)(0x0)
1921    #define TPM_NT_COUNTER      (TPM_NT)(0x1)
1922    #define TPM_NT_BITS         (TPM_NT)(0x2)
1923    #define TPM_NT_EXTEND       (TPM_NT)(0x4)
1924    #define TPM_NT_PIN_FAIL     (TPM_NT)(0x8)
1925    #define TPM_NT_PIN_PASS     (TPM_NT)(0x9)
1926    typedef struct {                                    // Table 2:214
1927        UINT32                  pinCount;
1928        UINT32                  pinLimit;
1929    } TPMS_NV_PIN_COUNTER_PARAMETERS;                   /* Structure */
1930    #define TYPE_OF_TPMA_NV     UINT32
```

```
1931    #define TPMA_NV_TO_UINT32(a)      (*((UINT32 *)&(a)))
1932    #define UINT32_TO_TPMA_NV(a)      (*((TPMA_NV *)&(a)))
1933    #define TPMA_NV_TO_BYTE_ARRAY(i, a)                                          \
1934                UINT32_TO_BYTE_ARRAY((TPMA_NV_TO_UINT32(i)), (a))
1935    #define BYTE_ARRAY_TO_TPMA_NV(i, a)                                          \
1936                { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_NV(x); }
1937    #if USE_BIT_FIELD_STRUCTURES
1938    typedef struct TPMA_NV {                                  // Table 2:215
1939        unsigned    PPWRITE              : 1;
1940        unsigned    OWNERWRITE           : 1;
1941        unsigned    AUTHWRITE            : 1;
1942        unsigned    POLICYWRITE          : 1;
1943        unsigned    TPM_NT               : 4;
1944        unsigned    Reserved_bits_at_8   : 2;
1945        unsigned    POLICY_DELETE        : 1;
1946        unsigned    WRITELOCKED          : 1;
1947        unsigned    WRITEALL             : 1;
1948        unsigned    WRITEDEFINE          : 1;
1949        unsigned    WRITE_STCLEAR        : 1;
1950        unsigned    GLOBALLOCK           : 1;
1951        unsigned    PPREAD               : 1;
1952        unsigned    OWNERREAD            : 1;
1953        unsigned    AUTHREAD             : 1;
1954        unsigned    POLICYREAD           : 1;
1955        unsigned    Reserved_bits_at_20  : 5;
1956        unsigned    NO_DA                : 1;
1957        unsigned    ORDERLY              : 1;
1958        unsigned    CLEAR_STCLEAR        : 1;
1959        unsigned    READLOCKED           : 1;
1960        unsigned    WRITTEN              : 1;
1961        unsigned    PLATFORMCREATE       : 1;
1962        unsigned    READ_STCLEAR         : 1;
1963    } TPMA_NV;                                                /* Bits */
```

This is the initializer for a TPMA_NV structure

```
1964    #define TPMA_NV_INITIALIZER(                                                 \
1965                 ppwrite,        ownerwrite,     authwrite,      policywrite,    \
1966                 tpm_nt,         bits_at_8,      policy_delete,  writelocked,    \
1967                 writeall,       writedefine,    write_stclear,  globallock,     \
1968                 ppread,         ownerread,      authread,       policyread,     \
1969                 bits_at_20,     no_da,          orderly,        clear_stclear,  \
1970                 readlocked,     written,        platformcreate, read_stclear)   \
1971                {ppwrite,        ownerwrite,     authwrite,      policywrite,    \
1972                 tpm_nt,         bits_at_8,      policy_delete,  writelocked,    \
1973                 writeall,       writedefine,    write_stclear,  globallock,     \
1974                 ppread,         ownerread,      authread,       policyread,     \
1975                 bits_at_20,     no_da,          orderly,        clear_stclear,  \
1976                 readlocked,     written,        platformcreate, read_stclear}
1977    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:215 TPMA_NV using bit masking

```
1978    typedef UINT32                  TPMA_NV;
1979    #define TYPE_OF_TPMA_NV         UINT32
1980    #define TPMA_NV_PPWRITE         ((TPMA_NV)1 << 0)
1981    #define TPMA_NV_OWNERWRITE      ((TPMA_NV)1 << 1)
1982    #define TPMA_NV_AUTHWRITE       ((TPMA_NV)1 << 2)
1983    #define TPMA_NV_POLICYWRITE     ((TPMA_NV)1 << 3)
1984    #define TPMA_NV_TPM_NT_SHIFT    4
1985    #define TPMA_NV_TPM_NT          ((TPMA_NV)0xf << 4)
1986    #define TPMA_NV_POLICY_DELETE   ((TPMA_NV)1 << 10)
1987    #define TPMA_NV_WRITELOCKED     ((TPMA_NV)1 << 11)
1988    #define TPMA_NV_WRITEALL        ((TPMA_NV)1 << 12)
1989    #define TPMA_NV_WRITEDEFINE     ((TPMA_NV)1 << 13)
```

```
1990    #define TPMA_NV_WRITE_STCLEAR   ((TPMA_NV)1 << 14)
1991    #define TPMA_NV_GLOBALLOCK      ((TPMA_NV)1 << 15)
1992    #define TPMA_NV_PPREAD          ((TPMA_NV)1 << 16)
1993    #define TPMA_NV_OWNERREAD       ((TPMA_NV)1 << 17)
1994    #define TPMA_NV_AUTHREAD        ((TPMA_NV)1 << 18)
1995    #define TPMA_NV_POLICYREAD      ((TPMA_NV)1 << 19)
1996    #define TPMA_NV_NO_DA           ((TPMA_NV)1 << 25)
1997    #define TPMA_NV_ORDERLY         ((TPMA_NV)1 << 26)
1998    #define TPMA_NV_CLEAR_STCLEAR   ((TPMA_NV)1 << 27)
1999    #define TPMA_NV_READLOCKED      ((TPMA_NV)1 << 28)
2000    #define TPMA_NV_WRITTEN         ((TPMA_NV)1 << 29)
2001    #define TPMA_NV_PLATFORMCREATE  ((TPMA_NV)1 << 30)
2002    #define TPMA_NV_READ_STCLEAR    ((TPMA_NV)1 << 31)
```

This is the initializer for a TPMA_NV bit array.

```
2003    #define TPMA_NV_INITIALIZER(                                                    \
2004                ppwrite,        ownerwrite,     authwrite,      policywrite,        \
2005                tpm_nt,         bits_at_8,      policy_delete,  writelocked,        \
2006                writeall,       writedefine,    write_stclear,  globallock,         \
2007                ppread,         ownerread,      authread,       policyread,         \
2008                bits_at_20,     no_da,          orderly,        clear_stclear,      \
2009                readlocked,     written,        platformcreate, read_stclear)       \
2010               {(ppwrite << 0)         + (ownerwrite << 1)      +                   \
2011                (authwrite << 2)       + (policywrite << 3)     +                   \
2012                (tpm_nt << 4)          + (policy_delete << 10)  +                   \
2013                (writelocked << 11)    + (writeall << 12)       +                   \
2014                (writedefine << 13)    + (write_stclear << 14)  +                   \
2015                (globallock << 15)     + (pppread << 16)        +                   \
2016                (ownerread << 17)      + (authread << 18)       +                   \
2017                (policyread << 19)     + (no_da << 25)          +                   \
2018                (orderly << 26)        + (clear_stclear << 27)  +                   \
2019                (readlocked << 28)     + (written << 29)        +                   \
2020                (platformcreate << 30) + (read_stclear << 31)}
2021    #endif // USE_BIT_FIELD_STRUCTURES
2022    typedef struct {                                        // Table 2:216
2023        TPMI_RH_NV_INDEX        nvIndex;
2024        TPMI_ALG_HASH           nameAlg;
2025        TPMA_NV                 attributes;
2026        TPM2B_DIGEST            authPolicy;
2027        UINT16                  dataSize;
2028    } TPMS_NV_PUBLIC;                                       /* Structure */
2029    typedef struct {                                        // Table 2:217
2030        UINT16                  size;
2031        TPMS_NV_PUBLIC          nvPublic;
2032    } TPM2B_NV_PUBLIC;                                      /* Structure */
2033    typedef union {                                         // Table 2:218
2034        struct {
2035            UINT16              size;
2036            BYTE                buffer[MAX_CONTEXT_SIZE];
2037        }           t;
2038        TPM2B       b;
2039    } TPM2B_CONTEXT_SENSITIVE;                              /* Structure */
2040    typedef struct {                                        // Table 2:219
2041        TPM2B_DIGEST                    integrity;
2042        TPM2B_CONTEXT_SENSITIVE         encrypted;
2043    } TPMS_CONTEXT_DATA;                                    /* Structure */
2044    typedef union {                                         // Table 2:220
2045        struct {
2046            UINT16              size;
2047            BYTE                buffer[sizeof(TPMS_CONTEXT_DATA)];
2048        }           t;
2049        TPM2B       b;
2050    } TPM2B_CONTEXT_DATA;                                   /* Structure */
2051    typedef struct {                                        // Table 2:221
```

```
2052        UINT64                  sequence;
2053        TPMI_DH_SAVED           savedHandle;
2054        TPMI_RH_HIERARCHY       hierarchy;
2055        TPM2B_CONTEXT_DATA      contextBlob;
2056    } TPMS_CONTEXT;                                     /* Structure */
2057    typedef struct {                                    // Table 2:223
2058        TPML_PCR_SELECTION      pcrSelect;
2059        TPM2B_DIGEST            pcrDigest;
2060        TPMA_LOCALITY           locality;
2061        TPM_ALG_ID              parentNameAlg;
2062        TPM2B_NAME              parentName;
2063        TPM2B_NAME              parentQualifiedName;
2064        TPM2B_DATA              outsideInfo;
2065    } TPMS_CREATION_DATA;                               /* Structure */
2066    typedef struct {                                    // Table 2:224
2067        UINT16                  size;
2068        TPMS_CREATION_DATA      creationData;
2069    } TPM2B_CREATION_DATA;                              /* Structure */
```

Table 2:225 - Definition of TPM_AT Constants

```
2070    typedef UINT32              TPM_AT;
2071    #define TYPE_OF_TPM_AT      UINT32
2072    #define TPM_AT_ANY          (TPM_AT)(0x00000000)
2073    #define TPM_AT_ERROR        (TPM_AT)(0x00000001)
2074    #define TPM_AT_PV1          (TPM_AT)(0x00000002)
2075    #define TPM_AT_VEND         (TPM_AT)(0x80000000)
```

Table 2:226 - Definition of TPM_AE Constants

```
2076    typedef UINT32              TPM_AE;
2077    #define TYPE_OF_TPM_AE      UINT32
2078    #define TPM_AE_NONE         (TPM_AE)(0x00000000)
2079    typedef struct {                                    // Table 2:227
2080        TPM_AT                  tag;
2081        UINT32                  data;
2082    } TPMS_AC_OUTPUT;                                   /* Structure */
2083    typedef struct {                                    // Table 2:228
2084        UINT32                  count;
2085        TPMS_AC_OUTPUT          acCapabilities[MAX_AC_CAPABILITIES];
2086    } TPML_AC_CAPABILITIES;                             /* Structure */
2087    #endif // _TPM_TYPES_H_
```

5.20 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
```

5.21 swap.h

```
#ifndef _SWAP_H
#define _SWAP_H
#if LITTLE_ENDIAN_TPM
#define TO_BIG_ENDIAN_UINT16(i)     REVERSE_ENDIAN_16(i)
#define FROM_BIG_ENDIAN_UINT16(i)   REVERSE_ENDIAN_16(i)
#define TO_BIG_ENDIAN_UINT32(i)     REVERSE_ENDIAN_32(i)
#define FROM_BIG_ENDIAN_UINT32(i)   REVERSE_ENDIAN_32(i)
#define TO_BIG_ENDIAN_UINT64(i)     REVERSE_ENDIAN_64(i)
#define FROM_BIG_ENDIAN_UINT64(i)   REVERSE_ENDIAN_64(i)
#else
#define TO_BIG_ENDIAN_UINT16(i)     (i)
#define FROM_BIG_ENDIAN_UINT16(i)   (i)
#define TO_BIG_ENDIAN_UINT32(i)     (i)
#define FROM_BIG_ENDIAN_UINT32(i)   (i)
#define TO_BIG_ENDIAN_UINT64(i)     (i)
#define FROM_BIG_ENDIAN_UINT64(i)   (i)
#endif
#if   AUTO_ALIGN == NO
```

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

```
#define BYTE_ARRAY_TO_UINT8(b)   (uint8_t)((b)[0])
#define BYTE_ARRAY_TO_UINT16(b) ByteArrayToUint16((BYTE *)(b))
#define BYTE_ARRAY_TO_UINT32(b) ByteArrayToUint32((BYTE *)(b))
#define BYTE_ARRAY_TO_UINT64(b) ByteArrayToUint64((BYTE *)(b))
#define UINT8_TO_BYTE_ARRAY(i, b) ((b)[0] = (uint8_t)(i))
#define UINT16_TO_BYTE_ARRAY(i, b)  Uint16ToByteArray((i), (BYTE *)(b))
#define UINT32_TO_BYTE_ARRAY(i, b)  Uint32ToByteArray((i), (BYTE *)(b))
#define UINT64_TO_BYTE_ARRAY(i, b)  Uint64ToByteArray((i), (BYTE *)(b))
#else // AUTO_ALIGN
#if BIG_ENDIAN_TPM
```

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

```
#define BYTE_ARRAY_TO_UINT8(b)        *((uint8_t  *)(b))
#define BYTE_ARRAY_TO_UINT16(b)       *((uint16_t *)(b))
#define BYTE_ARRAY_TO_UINT32(b)       *((uint32_t *)(b))
#define BYTE_ARRAY_TO_UINT64(b)       *((uint64_t *)(b))
```

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)   {*((uint8_t  *)(b)) = (i);}
#define UINT16_TO_BYTE_ARRAY(i, b)  {*((uint16_t *)(b)) = (i);}
#define UINT32_TO_BYTE_ARRAY(i, b)  {*((uint32_t *)(b)) = (i);}
#define UINT64_TO_BYTE_ARRAY(i, b)  {*((uint64_t *)(b)) = (i);}
#else
```

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

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

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)   {*((uint8_t  *)(b)) = (i);}
```

```
43    #define UINT16_TO_BYTE_ARRAY(i, b)  {*((uint16_t *)(b)) = REVERSE_ENDIAN_16(i);}
44    #define UINT32_TO_BYTE_ARRAY(i, b)  {*((uint32_t *)(b)) = REVERSE_ENDIAN_32(i);}
45    #define UINT64_TO_BYTE_ARRAY(i, b)  {*((uint64_t *)(b)) = REVERSE_ENDIAN_64(i);}
46    #endif   // BIG_ENDIAN_TPM
47    #endif  // AUTO_ALIGN == NO
48    #endif  // _SWAP_H
```

5.22 ACT.h

```
#ifndef _ACT_H_
#define _ACT_H_
#include "TpmProfile.h"
#if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
#   undef   RH_ACT_0
#   define  RH_ACT_0 NO
#   define IF_ACT_0_IMPLEMENTED(op)
#else
#   define IF_ACT_0_IMPLEMENTED(op) op(0)
#endif
#if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
#   undef   RH_ACT_1
#   define  RH_ACT_1 NO
#   define IF_ACT_1_IMPLEMENTED(op)
#else
#   define IF_ACT_1_IMPLEMENTED(op) op(1)
#endif
#if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
#   undef   RH_ACT_2
#   define  RH_ACT_2 NO
#   define IF_ACT_2_IMPLEMENTED(op)
#else
#   define IF_ACT_2_IMPLEMENTED(op) op(2)
#endif
#if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
#   undef   RH_ACT_3
#   define  RH_ACT_3 NO
#   define IF_ACT_3_IMPLEMENTED(op)
#else
#   define IF_ACT_3_IMPLEMENTED(op) op(3)
#endif
#if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
#   undef   RH_ACT_4
#   define  RH_ACT_4 NO
#   define IF_ACT_4_IMPLEMENTED(op)
#else
#   define IF_ACT_4_IMPLEMENTED(op) op(4)
#endif
#if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
#   undef   RH_ACT_5
#   define  RH_ACT_5 NO
#   define IF_ACT_5_IMPLEMENTED(op)
#else
#   define IF_ACT_5_IMPLEMENTED(op) op(5)
#endif
#if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
#   undef   RH_ACT_6
#   define  RH_ACT_6 NO
#   define IF_ACT_6_IMPLEMENTED(op)
#else
#   define IF_ACT_6_IMPLEMENTED(op) op(6)
#endif
#if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
#   undef   RH_ACT_7
#   define  RH_ACT_7 NO
#   define IF_ACT_7_IMPLEMENTED(op)
#else
#   define IF_ACT_7_IMPLEMENTED(op) op(7)
#endif
#if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
#   undef   RH_ACT_8
#   define  RH_ACT_8 NO
#   define IF_ACT_8_IMPLEMENTED(op)
```

```
64  #else
65  #   define IF_ACT_8_IMPLEMENTED(op) op(8)
66  #endif
67  #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
68  #   undef   RH_ACT_9
69  #   define  RH_ACT_9 NO
70  #   define IF_ACT_9_IMPLEMENTED(op)
71  #else
72  #   define IF_ACT_9_IMPLEMENTED(op) op(9)
73  #endif
74  #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
75  #   undef   RH_ACT_A
76  #   define  RH_ACT_A NO
77  #   define IF_ACT_A_IMPLEMENTED(op)
78  #else
79  #   define IF_ACT_A_IMPLEMENTED(op) op(A)
80  #endif
81  #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
82  #   undef   RH_ACT_B
83  #   define  RH_ACT_B NO
84  #   define IF_ACT_B_IMPLEMENTED(op)
85  #else
86  #   define IF_ACT_B_IMPLEMENTED(op) op(B)
87  #endif
88  #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
89  #   undef   RH_ACT_C
90  #   define  RH_ACT_C NO
91  #   define IF_ACT_C_IMPLEMENTED(op)
92  #else
93  #   define IF_ACT_C_IMPLEMENTED(op) op(C)
94  #endif
95  #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
96  #   undef   RH_ACT_D
97  #   define  RH_ACT_D NO
98  #   define IF_ACT_D_IMPLEMENTED(op)
99  #else
100 #   define IF_ACT_D_IMPLEMENTED(op) op(D)
101 #endif
102 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
103 #   undef   RH_ACT_E
104 #   define  RH_ACT_E NO
105 #   define IF_ACT_E_IMPLEMENTED(op)
106 #else
107 #   define IF_ACT_E_IMPLEMENTED(op) op(E)
108 #endif
109 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
110 #   undef   RH_ACT_F
111 #   define  RH_ACT_F NO
112 #   define IF_ACT_F_IMPLEMENTED(op)
113 #else
114 #   define IF_ACT_F_IMPLEMENTED(op) op(F)
115 #endif
116 #ifndef TPM_RH_ACT_0
117 #error Need numeric definition for TPM_RH_ACT_0
118 #endif
119 #ifndef TPM_RH_ACT_1
120 #   define TPM_RH_ACT_1    (TPM_RH_ACT_0 + 1)
121 #endif
122 #ifndef TPM_RH_ACT_2
123 #   define TPM_RH_ACT_2    (TPM_RH_ACT_0 + 2)
124 #endif
125 #ifndef TPM_RH_ACT_3
126 #   define TPM_RH_ACT_3    (TPM_RH_ACT_0 + 3)
127 #endif
128 #ifndef TPM_RH_ACT_4
129 #   define TPM_RH_ACT_4    (TPM_RH_ACT_0 + 4)
```

```
#endif
#ifndef TPM_RH_ACT_5
#   define TPM_RH_ACT_5    (TPM_RH_ACT_0 + 5)
#endif
#ifndef TPM_RH_ACT_6
#   define TPM_RH_ACT_6    (TPM_RH_ACT_0 + 6)
#endif
#ifndef TPM_RH_ACT_7
#   define TPM_RH_ACT_7    (TPM_RH_ACT_0 + 7)
#endif
#ifndef TPM_RH_ACT_8
#   define TPM_RH_ACT_8    (TPM_RH_ACT_0 + 8)
#endif
#ifndef TPM_RH_ACT_9
#   define TPM_RH_ACT_9    (TPM_RH_ACT_0 + 9)
#endif
#ifndef TPM_RH_ACT_A
#   define TPM_RH_ACT_A    (TPM_RH_ACT_0 + 0xA)
#endif
#ifndef TPM_RH_ACT_B
#   define TPM_RH_ACT_B    (TPM_RH_ACT_0 + 0xB)
#endif
#ifndef TPM_RH_ACT_C
#   define TPM_RH_ACT_C    (TPM_RH_ACT_0 + 0xC)
#endif
#ifndef TPM_RH_ACT_D
#   define TPM_RH_ACT_D    (TPM_RH_ACT_0 + 0xD)
#endif
#ifndef TPM_RH_ACT_E
#   define TPM_RH_ACT_E    (TPM_RH_ACT_0 + 0xE)
#endif
#ifndef TPM_RH_ACT_F
#   define TPM_RH_ACT_F    (TPM_RH_ACT_0 + 0xF)
#endif
#define FOR_EACH_ACT(op)                                                    \
    IF_ACT_0_IMPLEMENTED(op)                                                \
    IF_ACT_1_IMPLEMENTED(op)                                                \
    IF_ACT_2_IMPLEMENTED(op)                                                \
    IF_ACT_3_IMPLEMENTED(op)                                                \
    IF_ACT_4_IMPLEMENTED(op)                                                \
    IF_ACT_5_IMPLEMENTED(op)                                                \
    IF_ACT_6_IMPLEMENTED(op)                                                \
    IF_ACT_7_IMPLEMENTED(op)                                                \
    IF_ACT_8_IMPLEMENTED(op)                                                \
    IF_ACT_9_IMPLEMENTED(op)                                                \
    IF_ACT_A_IMPLEMENTED(op)                                                \
    IF_ACT_B_IMPLEMENTED(op)                                                \
    IF_ACT_C_IMPLEMENTED(op)                                                \
    IF_ACT_D_IMPLEMENTED(op)                                                \
    IF_ACT_E_IMPLEMENTED(op)                                                \
    IF_ACT_F_IMPLEMENTED(op)
```

This is the mask for ACT that are implemented

```
//#define ACT_MASK(N)      | (1 << 0x##N)
//#define ACT_IMPLEMENTED_MASK    (0 FOR_EACH_ACT(ACT_MASK))
#define CASE_ACT_HANDLE(N)      case TPM_RH_ACT_##N:
#define CASE_ACT_NUMBER(N)      case 0x##N:
typedef struct ACT_STATE
{
    UINT32          remaining;
    TPM_ALG_ID      hashAlg;
    TPM2B_DIGEST    authPolicy;
} ACT_STATE, *P_ACT_STATE;
#endif // _ACT_H_
```

6 Main

6.1 Introduction

The files in this section are the main processing blocks for the TPM. ExecuteCommand.c contains the entry point into the TPM code and the parsing of the command header. SessionProcess.c handles the parsing of the session area and the authorization checks, and CommandDispatch.c does the parameter unmarshaling and command dispatch.

6.2 ExecCommand.c

6.2.1 Introduction

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

6.2.2 Includes

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

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

```
3  // #include "CommandResponseSizes_fp.h"
```

6.2.3 ExecuteCommand()

The function performs the following steps.

a) Parses the command header from input buffer.

b) Calls ParseHandleBuffer() to parse the handle area of the command.

c) Validates that each of the handles references a loaded entity.

d) Calls ParseSessionBuffer() () to:

   1) unmarshal and parse the session area;

   2) check the authorizations; and

   3) when necessary, decrypt a parameter.

e) Calls CommandDispatcher() to:

   1) unmarshal the command parameters from the command buffer;

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

   3) marshal the responses into the response buffer.

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

g) Calls BuildResponseSessions() to:

   1) when necessary, encrypt a parameter

   2) build the response authorization sessions

   3) update the audit sessions and nonces

h) Calls BuildResponseHeader() to complete the construction of the response.

responseSize is set by the caller to the maximum number of bytes available in the output buffer. ExecuteCommand() will adjust the value and return the number of bytes placed in the buffer.

response is also set by the caller to indicate the buffer into which ExecuteCommand() is to place the response.

request and *response* may point to the same buffer

NOTE: As of February, 2016, the failure processing has been moved to the platform-specific code. When the TPM code encounters an unrecoverable failure, it will SET *g_inFailureMode* and call _plat__Fail(). That function should not return but may call ExecuteCommand().

```
LIB_EXPORT void
ExecuteCommand(
    uint32_t         requestSize,   // IN: command buffer size
    unsigned char   *request,       // IN: command buffer
    uint32_t        *responseSize,  // IN/OUT: response buffer size
    unsigned char   **response      // IN/OUT: response buffer
    )
{
    // Command local variables
    UINT32               commandSize;
    COMMAND              command;

    // Response local variables
    UINT32               maxResponse = *responseSize;
    TPM_RC               result;            // return code for the command

// This next function call is used in development to size the command and response
// buffers. The values printed are the sizes of the internal structures and
// not the sizes of the canonical forms of the command response structures. Also,
// the sizes do not include the tag, command.code, requestSize, or the authorization
// fields.
//CommandResponseSizes();
    // Set flags for NV access state. This should happen before any other
    // operation that may require a NV write. Note, that this needs to be done
```

```
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"
#include "Marshal.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_t
{
    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.

```
20  typedef struct COMMAND_DESCRIPTOR_t
21  {
22      COMMAND_t       command;        // Address of the command
23      UINT16          inSize;         // Maximum size of the input structure
24      UINT16          outSize;        // Maximum size of the output structure
25      UINT16          typesOffset;    // address of the types field
26      UINT16          offsets[1];
27  } 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
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;

```

```
67          // Get the first type value
68          for(type = *types++;
69              // check each byte to make sure that we have not hit the start
70              // of the parameters
71          (dType = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
72          // get the next type
73              type = *types++)
74          {
75      #if TABLE_DRIVEN_MARSHAL
76              marshalIndex_t      index;
77              index = UnmarshalArray[dType] | ((type & 0x80) ? NULL_FLAG : 0);
78              result = Unmarshal(index, &(command->handles[command->handleNum]),
79                                 &command->parameterBuffer, &command->parameterSize);
80
81      #else
82              // See if unmarshaling of this handle type requires a flag
83              if(dType < HANDLE_FIRST_FLAG_TYPE)
84              {
85                  // Look up the function to do the unmarshaling
86                  NoFlagFunction  *f = (NoFlagFunction *)UnmarshalArray[dType];
87                  // call it
88                  result = f(&(command->handles[command->handleNum]),
89                             &command->parameterBuffer,
90                             &command->parameterSize);
91              }
92              else
93              {
94                  //  Look up the function
95                  FlagFunction    *f = UnmarshalArray[dType];
96
97                  // Call it setting the flag to the appropriate value
98                  result = f(&(command->handles[command->handleNum]),
99                             &command->parameterBuffer,
100                            &command->parameterSize, (type & 0x80) != 0);
101             }
102     #endif
103
104             // Got a handle
105             // We do this first so that the match for the handle offset of the
106             // response code works correctly.
107             command->handleNum += 1;
108             if(result != TPM_RC_SUCCESS)
109                 // if the unmarshaling failed, return the response code with the
110                 // handle indication set
111                 return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
112         }
113     #else
114         BYTE            **handleBufferStart = &command->parameterBuffer;
115         INT32           *bufferRemainingSize = &command->parameterSize;
116         TPM_HANDLE      *handles = &command->handles[0];
117         UINT32          *handleCount = &command->handleNum;
118         *handleCount = 0;
119         switch(command->code)
120         {
121     #include "HandleProcess.h"
122     #undef handles
123             default:
124                 FAIL(FATAL_ERROR_INTERNAL);
125                 break;
126         }
127     #endif
128         return TPM_RC_SUCCESS;
129     }
```

6.3.1.2.2 CommandDispatcher()

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

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

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

```
256             goto Exit;
257         }
258         // This check is used so that we don't have to add an additional offset
259         // value to the offsets list to correspond to the stop value in the
260         // command parameter list.
261         if(*types != 0xFF)
262             offset = *offsets++;
263         hasInParameters++;
264     }
265     // Should have used all the bytes in the input
266     if(command->parameterSize != 0)
267     {
268         result = TPM_RC_SIZE;
269         goto Exit;
270     }
271
272     // The command parameter unmarshaling stopped when it hit a value that was out
273     // of range for unmarshaling values and left *types pointing to the first
274     // marshaling type. If that type happens to be the STOP value, then there
275     // are no response parameters. So, set the flag to indicate if there are
276     // output parameters.
277     hasOutParameters = *types != 0xFF;
278
279     // There are four cases for calling, with and without input parameters and with
280     // and without output parameters.
281     if(hasInParameters > 0)
282     {
283         if(hasOutParameters)
284             result = cmd.inOutArg(commandIn, commandOut);
285         else
286             result = cmd.inArg(commandIn);
287     }
288     else
289     {
290         if(hasOutParameters)
291             result = cmd.outArg(commandOut);
292         else
293             result = cmd.noArgs();
294     }
295     if(result != TPM_RC_SUCCESS)
296        goto Exit;
297
298     // Offset in the marshaled output structure
299     offset = 0;
300
301     // Process the return handles, if any
302     command->handleNum = 0;
303
304     // Could make this a loop to process output handles but there is only ever
305     // one handle in the outputs (for now).
306     type = *types++;
307     if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
308     {
309         // The out->handle value was referenced as TPM_HANDLE in the
310         // action code so it has to be properly aligned.
311         command->handles[command->handleNum++] =
312             *((TPM_HANDLE *)&(commandOut[offset]));
313         maxOutSize -= sizeof(UINT32);
314         type = *types++;
315         offset = *offsets++;
316     }
317     // Use the size of the command action output buffer as the maximum for the
318     // number of bytes that can get marshaled. Since the marshaling code has
319     // no pointers to data, all of the data being returned has to be in the
320     // command action output buffer. If we try to marshal more bytes than
321     // could fit into the output buffer, we need to fail.
```

```
322         for(;(dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
323             && !g_inFailureMode; type = *types++)
324         {
325     #if TABLE_DRIVEN_MARSHAL
326             marshalIndex_t      index = MarshalArray[dType];
327             command->parameterSize += Marshal(index, &commandOut[offset],
328                                               &command->responseBuffer,
329                                               &maxOutSize);
330     #else
331             const MARSHAL_t     f = MarshalArray[dType];
332
333             command->parameterSize += f(&commandOut[offset],
334                                         &command->responseBuffer,
335                                         &maxOutSize);
336     #endif
337             offset = *offsets++;
338         }
339         result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
340     Exit:
341         MemoryIoBufferZero();
342         return result;
343     #endif
344     }
```

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"
#include "ACT.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;
```

```
        }
        case TPM_HT_PCR:
            // PCRs are always exempted from DA.
            result = TRUE;
            break;
        default:
            break;
    }
    return result;
}
```

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

```
static TPM_RC
IncrementLockout(
    UINT32              sessionIndex
    )
{
    TPM_HANDLE          handle = s_associatedHandles[sessionIndex];
    TPM_HANDLE          sessionHandle = s_sessionHandles[sessionIndex];
    SESSION            *session = NULL;
//
    // Don't increment lockout unless the handle associated with the session
    // is DA protected or the session is bound to a DA protected entity.
    if(sessionHandle == TPM_RS_PW)
    {
        if(IsDAExempted(handle))
            return TPM_RC_BAD_AUTH;
    }
    else
    {
        session = SessionGet(sessionHandle);
        // If the session is bound to lockout, then use that as the relevant
        // handle. This means that an authorization failure with a bound session
        // bound to lockoutAuth will take precedence over any other
        // lockout check
        if(session->attributes.isLockoutBound == SET)
            handle = TPM_RH_LOCKOUT;
        if(session->attributes.isDaBound == CLEAR
           && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
           // If the handle was changed to TPM_RH_LOCKOUT, this will not return
           // TPM_RC_BAD_AUTH
            return TPM_RC_BAD_AUTH;
    }
    if(handle == TPM_RH_LOCKOUT)
    {
        pAssert(gp.lockOutAuthEnabled == TRUE);

        // lockout is no longer enabled
        gp.lockOutAuthEnabled = FALSE;

        // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
        // the lockout authorization will be reset at startup.
        if(gp.lockoutRecovery != 0)
```

```
82          {
83              if(NV_IS_AVAILABLE)
84                  // Update NV.
85                  NV_SYNC_PERSISTENT(lockOutAuthEnabled);
86              else
87                  // No NV access for now. Put the TPM in pending mode.
88                  s_DAPendingOnNV = TRUE;
89          }
90      }
91      else
92      {
93          if(gp.recoveryTime != 0)
94          {
95              gp.failedTries++;
96              if(NV_IS_AVAILABLE)
97                  // Record changes to NV. NvWrite will SET g_updateNV
98                  NV_SYNC_PERSISTENT(failedTries);
99              else
100                 // No NV access for now.  Put the TPM in pending mode.
101                 s_DAPendingOnNV = TRUE;
102         }
103     }
104     // Register a DA failure and reset the timers.
105     DARegisterFailure(handle);
106
107     return TPM_RC_AUTH_FAIL;
108 }
```

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

```
109 static BOOL
110 IsSessionBindEntity(
111     TPM_HANDLE       associatedHandle,  // IN: handle to be authorized
112     SESSION         *session            // IN: associated session
113     )
114 {
115     TPM2B_NAME     entity;             // The bind value for the entity
116 //
117     // If the session is not bound, return FALSE.
118     if(session->attributes.isBound)
119     {
120         // Compute the bind value for the entity.
121         SessionComputeBoundEntity(associatedHandle, &entity);
122
123         // Compare to the bind value in the session.
124         return MemoryEqual2B(&entity.b, &session->u1.boundEntity.b);
125     }
126     return FALSE;
127 }
```

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

```
128  static BOOL
129  IsPolicySessionRequired(
130      COMMAND_INDEX    commandIndex,  // IN: command index
131      UINT32           sessionIndex   // IN: session index
132      )
133  {
134      AUTH_ROLE       role = CommandAuthRole(commandIndex, sessionIndex);
135      TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
136  //
137      if(role == AUTH_DUP)
138          return TRUE;
139      if(role == AUTH_ADMIN)
140      {
141          // We allow an exception for ADMIN role in a transient object. If the object
142          // allows ADMIN role actions with authorization, then policy is not
143          // required. For all other cases, there is no way to override the command
144          // requirement that a policy be used
145          if(type == TPM_HT_TRANSIENT)
146          {
147              OBJECT      *object = HandleToObject(s_associatedHandles[sessionIndex]);
148
149              if(!IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
150                               adminWithPolicy))
151                  return FALSE;
152          }
153          return TRUE;
154      }
155
156      if(type == TPM_HT_PCR)
157      {
158          if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
159          {
160              TPM2B_DIGEST        policy;
161              TPMI_ALG_HASH       policyAlg;
162              policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
163                                           &policy);
164              if(policyAlg != TPM_ALG_NULL)
165                  return TRUE;
166          }
167      }
168      return FALSE;
169  }
```

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

```
170  static BOOL
171  IsAuthValueAvailable(
172      TPM_HANDLE       handle,        // IN: handle of entity
173      COMMAND_INDEX    commandIndex,  // IN: command index
174      UINT32           sessionIndex   // IN: session index
175      )
176  {
177      BOOL             result = FALSE;
178  //
179      switch(HandleGetType(handle))
180      {
181          case TPM_HT_PERMANENT:
182              switch(handle)
183              {
184                      // At this point hierarchy availability has already been
185                      // checked so primary seed handles are always available here
186                  case TPM_RH_OWNER:
187                  case TPM_RH_ENDORSEMENT:
188                  case TPM_RH_PLATFORM:
189  #ifdef VENDOR_PERMANENT
190                      // This vendor defined handle associated with the
191                      // manufacturer's shared secret
192                  case VENDOR_PERMANENT:
193  #endif
194                      // The DA checking has been performed on LockoutAuth but we
195                      // bypass the DA logic if we are using lockout policy. The
196                      // policy would allow execution to continue an lockoutAuth
197                      // could be used, even if direct use of lockoutAuth is disabled
198                  case TPM_RH_LOCKOUT:
199                      // NullAuth is always available.
200                  case TPM_RH_NULL:
201                      result = TRUE;
202                      break;
203                  FOR_EACH_ACT(CASE_ACT_HANDLE)
204                  {
205                      // The ACT auth value is not available if the platform is disabled
206                       result = g_phEnable == SET;
207                       break;
208                  }
209                  default:
210                      // Otherwise authValue is not available.
211                      break;
212              }
213              break;
214          case TPM_HT_TRANSIENT:
215              // A persistent object has already been loaded and the internal
216              // handle changed.
217          {
```

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

```
284                 // PCR handle.
285                 // authValue is always allowed for PCR
286                 result = TRUE;
287                 break;
288             default:
289                 // Otherwise, authValue is not available
290                 break;
291         }
292         return result;
293     }
```

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

```
294     static BOOL
295     IsAuthPolicyAvailable(
296         TPM_HANDLE       handle,        // IN: handle of entity
297         COMMAND_INDEX    commandIndex,  // IN: command index
298         UINT32           sessionIndex   // IN: session index
299         )
300     {
301         BOOL             result = FALSE;
302     //
303         switch(HandleGetType(handle))
304         {
305             case TPM_HT_PERMANENT:
306                 switch(handle)
307                 {
308                     // At this point hierarchy availability has already been checked.
309                     case TPM_RH_OWNER:
310                         if(gp.ownerPolicy.t.size != 0)
311                             result = TRUE;
312                         break;
313                     case TPM_RH_ENDORSEMENT:
314                         if(gp.endorsementPolicy.t.size != 0)
315                             result = TRUE;
316                         break;
317                     case TPM_RH_PLATFORM:
318                         if(gc.platformPolicy.t.size != 0)
319                             result = TRUE;
320                         break;
321     #define ACT_GET_POLICY(N)                                                   \
322                     case TPM_RH_ACT_##N:                                        \
323                         if(go.ACT_##N.authPolicy.t.size != 0)                   \
324                             result = TRUE;                                      \
325                         break;
326
327                     FOR_EACH_ACT(ACT_GET_POLICY)
328
329                     case TPM_RH_LOCKOUT:
330                         if(gp.lockoutPolicy.t.size != 0)
331                             result = TRUE;
332                         break;
333                     default:
```

```
                    break;
                }
                break;
            case TPM_HT_TRANSIENT:
            {
                // Object handle.
                // An evict object would already have been loaded and given a
                // transient object handle by this point.
                OBJECT  *object = HandleToObject(handle);
                // Policy authorization is not available for an object with only
                // public portion loaded.
                if(object->attributes.publicOnly == CLEAR)
                {
                    // Policy authorization is always available for an object but
                    // is never available for a sequence.
                    if(!ObjectIsSequence(object))
                        result = TRUE;
                }
                break;
            }
            case TPM_HT_NV_INDEX:
                // An NV Index.
            {
                NV_INDEX         *nvIndex = NvGetIndexInfo(handle, NULL);
                TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
//
                // If the policy size is not zero, check if policy can be used.
                if(nvIndex->publicArea.authPolicy.t.size != 0)
                {
                    // If policy session is required for this handle, always
                    // uses policy regardless of the attributes bit setting
                    if(IsPolicySessionRequired(commandIndex, sessionIndex))
                        result = TRUE;
                    // Otherwise, the presence of the policy depends on the NV
                    // attributes.
                    else if(IsWriteOperation(commandIndex))
                    {
                        if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
                            result = TRUE;
                    }
                    else
                    {
                        if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
                            result = TRUE;
                    }
                }
            }
            break;
            case TPM_HT_PCR:
                // PCR handle.
                if(PCRPolicyIsAvailable(handle))
                    result = TRUE;
                break;
            default:
                break;
        }
        return result;
    }
```

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

```
    void
```

```
393 ClearCpRpHashes(
394     COMMAND         *command
395     )
396 {
397 #if ALG_SHA1
398     command->sha1CpHash.t.size = 0;
399     command->sha1RpHash.t.size = 0;
400 #endif
401 #if ALG_SHA256
402     command->sha256CpHash.t.size = 0;
403     command->sha256RpHash.t.size = 0;
404 #endif
405 #if ALG_SHA384
406     command->sha384CpHash.t.size = 0;
407     command->sha384RpHash.t.size = 0;
408 #endif
409 #if ALG_SHA512
410     command->sha512CpHash.t.size = 0;
411     command->sha512RpHash.t.size = 0;
412 #endif
413 #if ALG_SM3_256
414     command->sm3_256CpHash.t.size = 0;
415     command->sm3_256RpHash.t.size = 0;
416 #endif
417 }
```

6.4.4.2 GetCpHashPointer()

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

```
418 static TPM2B_DIGEST *
419 GetCpHashPointer(
420     COMMAND         *command,
421     TPMI_ALG_HASH    hashAlg
422     )
423 {
424     TPM2B_DIGEST     *retVal;
425 //
426     switch(hashAlg)
427     {
428 #if ALG_SHA1
429         case ALG_SHA1_VALUE:
430             retVal = (TPM2B_DIGEST *)&command->sha1CpHash;
431             break;
432 #endif
433 #if ALG_SHA256
434         case ALG_SHA256_VALUE:
435             retVal = (TPM2B_DIGEST *)&command->sha256CpHash;
436             break;
437 #endif
438 #if ALG_SHA384
439         case ALG_SHA384_VALUE:
440             retVal = (TPM2B_DIGEST *)&command->sha384CpHash;
441             break;
442 #endif
443 #if ALG_SHA512
444         case ALG_SHA512_VALUE:
445             retVal = (TPM2B_DIGEST *)&command->sha512CpHash;
446             break;
447 #endif
448 #if ALG_SM3_256
449         case ALG_SM3_256_VALUE:
450             retVal = (TPM2B_DIGEST *)&command->sm3_256CpHash;
451             break;
```

```
452 #endif
453         default:
454             retVal = NULL;
455             break;
456     }
457     return retVal;
458 }
```

6.4.4.3 GetRpHashPointer()

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

```
459 static TPM2B_DIGEST *
460 GetRpHashPointer(
461     COMMAND         *command,
462     TPMI_ALG_HASH    hashAlg
463     )
464 {
465     TPM2B_DIGEST    *retVal;
466 //
467     switch(hashAlg)
468     {
469 #if ALG_SHA1
470         case ALG_SHA1_VALUE:
471             retVal = (TPM2B_DIGEST *)&command->sha1RpHash;
472             break;
473 #endif
474 #if ALG_SHA256
475         case ALG_SHA256_VALUE:
476             retVal = (TPM2B_DIGEST *)&command->sha256RpHash;
477             break;
478 #endif
479 #if ALG_SHA384
480         case ALG_SHA384_VALUE:
481             retVal = (TPM2B_DIGEST *)&command->sha384RpHash;
482             break;
483 #endif
484 #if ALG_SHA512
485         case ALG_SHA512_VALUE:
486             retVal = (TPM2B_DIGEST *)&command->sha512RpHash;
487             break;
488 #endif
489 #if ALG_SM3_256
490         case ALG_SM3_256_VALUE:
491             retVal = (TPM2B_DIGEST *)&command->sm3_256RpHash;
492             break;
493 #endif
494         default:
495             retVal = NULL;
496             break;
497     }
498     return retVal;
499 }
```

6.4.4.4 ComputeCpHash()

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

```
500 static TPM2B_DIGEST *
501 ComputeCpHash(
502     COMMAND         *command,       // IN: command parsing structure
503     TPMI_ALG_HASH    hashAlg        // IN: hash algorithm
504     )
```

```
505 {
506     UINT32               i;
507     HASH_STATE           hashState;
508     TPM2B_NAME           name;
509     TPM2B_DIGEST        *cpHash;
510 //
511     // cpHash = hash(commandCode [ || authName1
512     //                           [ || authName2
513     //                           [ || authName 3 ]]]
514     //                           [ || parameters])
515     // A cpHash can contain just a commandCode only if the lone session is
516     // an audit session.
517     // Get pointer to the hash value
518     cpHash = GetCpHashPointer(command, hashAlg);
519     if(cpHash->t.size == 0)
520     {
521         cpHash->t.size = CryptHashStart(&hashState, hashAlg);
522             //  Add commandCode.
523         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
524             //  Add authNames for each of the handles.
525         for(i = 0; i < command->handleNum; i++)
526             CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
527                                                            &name)->b);
528             //  Add the parameters.
529         CryptDigestUpdate(&hashState, command->parameterSize,
530                           command->parameterBuffer);
531             //  Complete the hash.
532         CryptHashEnd2B(&hashState, &cpHash->b);
533     }
534     return cpHash;
535 }
```

6.4.4.5 GetCpHash()

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

```
536 static TPM2B_DIGEST *
537 GetCpHash(
538     COMMAND         *command,
539     TPMI_ALG_HASH    hashAlg
540     )
541 {
542     TPM2B_DIGEST        *cpHash = GetCpHashPointer(command, hashAlg);
543  //
544     pAssert(cpHash->t.size != 0);
545     return cpHash;
546 }
```

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*

```
547 static BOOL
548 CompareTemplateHash(
549     COMMAND         *command,       // IN: parsing structure
```

```
550         SESSION         *session        // IN: session data
551         )
552     {
553         BYTE                *pBuffer = command->parameterBuffer;
554         INT32                pSize = command->parameterSize;
555         TPM2B_DIGEST         tHash;
556         UINT16               size;
557     //
558         // Only try this for the three commands for which it is intended
559         if(command->code != TPM_CC_Create
560            && command->code != TPM_CC_CreatePrimary
561     #if CC_CreateLoaded
562            && command->code != TPM_CC_CreateLoaded
563     #endif
564            )
565             return FALSE;
566         // Assume that the first parameter is a TPM2B and unmarshal the size field
567         // Note: this will not affect the parameter buffer and size in the calling
568         // function.
569         if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
570             return FALSE;
571         // reduce the space in the buffer.
572         // NOTE: this could make pSize go negative if the parameters are not correct but
573         // the unmarshaling code does not try to unmarshal if the remaining size is
574         // negative.
575         pSize -= size;
576
577         // Advance the pointer
578         pBuffer += size;
579
580         // Get the size of what should be the template
581         if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
582             return FALSE;
583         // See if this is reasonable
584         if(size > pSize)
585             return FALSE;
586         // Hash the template data
587         tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
588                                       sizeof(tHash.t.buffer), tHash.t.buffer);
589         return(MemoryEqual2B(&session->u1.templateHash.b, &tHash.b));
590     }
```

6.4.4.7 CompareNameHash()

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

```
591     BOOL
592     CompareNameHash(
593         COMMAND         *command,       // IN: main parsing structure
594         SESSION         *session        // IN: session structure with nameHash
595         )
596     {
597         HASH_STATE           hashState;
598         TPM2B_DIGEST         nameHash;
599         UINT32               i;
600         TPM2B_NAME           name;
601     //
602         nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
603         //  Add names.
604         for(i = 0; i < command->handleNum; i++)
605             CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
606                                                            &name)->b);
607         //  Complete hash.
608         CryptHashEnd2B(&hashState, &nameHash.b);
```

```
609         // and compare
610         return MemoryEqual(session->u1.nameHash.t.buffer, nameHash.t.buffer,
611                            nameHash.t.size);
612     }
```

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

```
613     static TPM_RC
614     CheckPWAuthSession(
615         UINT32           sessionIndex   // IN: index of session to be processed
616         )
617     {
618         TPM2B_AUTH      authValue;
619         TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
620     //
621         // Strip trailing zeros from the password.
622         MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
623
624         // Get the authValue with trailing zeros removed
625         EntityGetAuthValue(associatedHandle, &authValue);
626
627         // Success if the values are identical.
628         if(MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
629         {
630             return TPM_RC_SUCCESS;
631         }
632         else                    // if the digests are not identical
633         {
634             // Invoke DA protection if applicable.
635             return IncrementLockout(sessionIndex);
636         }
637     }
```

6.4.4.9 ComputeCommandHMAC()

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

```
638     static TPM2B_DIGEST *
639     ComputeCommandHMAC(
640         COMMAND         *command,       // IN: primary control structure
641         UINT32           sessionIndex,  // IN: index of session to be processed
642         TPM2B_DIGEST    *hmac           // OUT: authorization HMAC
643         )
644     {
645         TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
646         TPM2B_KEY        key;
647         BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
648         BYTE            *buffer;
649         UINT32           marshalSize;
650         HMAC_STATE       hmacState;
651         TPM2B_NONCE     *nonceDecrypt;
652         TPM2B_NONCE     *nonceEncrypt;
653         SESSION         *session;
```

```
654 //
655     nonceDecrypt = NULL;
656     nonceEncrypt = NULL;
657
658     // Determine if extra nonceTPM values are going to be required.
659     // If this is the first session (sessionIndex = 0) and it is an authorization
660     // session that uses an HMAC, then check if additional session nonces are to be
661     // included.
662     if(sessionIndex == 0
663        && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
664     {
665         // If there is a decrypt session and if this is not the decrypt session,
666         // then an extra nonce may be needed.
667         if(s_decryptSessionIndex != UNDEFINED_INDEX
668            && s_decryptSessionIndex != sessionIndex)
669         {
670             // Will add the nonce for the decrypt session.
671             SESSION *decryptSession
672                 = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
673             nonceDecrypt = &decryptSession->nonceTPM;
674         }
675         // Now repeat for the encrypt session.
676         if(s_encryptSessionIndex != UNDEFINED_INDEX
677            && s_encryptSessionIndex != sessionIndex
678            && s_encryptSessionIndex != s_decryptSessionIndex)
679         {
680             // Have to have the nonce for the encrypt session.
681             SESSION *encryptSession
682                 = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
683             nonceEncrypt = &encryptSession->nonceTPM;
684         }
685     }
686
687     // Continue with the HMAC processing.
688     session = SessionGet(s_sessionHandles[sessionIndex]);
689
690     // Generate HMAC key.
691     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
692
693     // Check if the session has an associated handle and if the associated entity
694     // is the one to which the session is bound. If not, add the authValue of
695     // this entity to the HMAC key.
696     // If the session is bound to the object or the session is a policy session
697     // with no authValue required, do not include the authValue in the HMAC key.
698     // Note: For a policy session, its isBound attribute is CLEARED.
699     //
700     // Include the entity authValue if it is needed
701     if(session->attributes.includeAuth == SET)
702     {
703         TPM2B_AUTH          authValue;
704         // Get the entity authValue with trailing zeros removed
705         EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
706         // add the authValue to the HMAC key
707         MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
708     }
709      // if the HMAC key size is 0, a NULL string HMAC is allowed
710     if(key.t.size == 0
711        && s_inputAuthValues[sessionIndex].t.size == 0)
712     {
713         hmac->t.size = 0;
714         return hmac;
715     }
716     // Start HMAC
717     hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
718
719         //  Add cpHash
```

```
720         CryptDigestUpdate2B(&hmacState.hashState,
721                             &ComputeCpHash(command, session->authHashAlg)->b);
722             //  Add nonces as required
723         CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
724         CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
725         if(nonceDecrypt != NULL)
726             CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
727         if(nonceEncrypt != NULL)
728             CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
729             //  Add sessionAttributes
730         buffer = marshalBuffer;
731         marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
732                                            &buffer, NULL);
733         CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
734             // Complete the HMAC computation
735         CryptHmacEnd2B(&hmacState, &hmac->b);
736
737         return hmac;
738     }
```

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

```
739     static TPM_RC
740     CheckSessionHMAC(
741         COMMAND         *command,       // IN: primary control structure
742         UINT32           sessionIndex   // IN: index of session to be processed
743         )
744     {
745         TPM2B_DIGEST        hmac;           // authHMAC for comparing
746     //
747         // Compute authHMAC
748         ComputeCommandHMAC(command, sessionIndex, &hmac);
749
750         // Compare the input HMAC with the authHMAC computed above.
751         if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
752         {
753             // If an HMAC session has a failure, invoke the anti-hammering
754             // if it applies to the authorized entity or the session.
755             // Otherwise, just indicate that the authorization is bad.
756             return IncrementLockout(sessionIndex);
757         }
758         return TPM_RC_SUCCESS;
759     }
```

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

```
760  static TPM_RC
761  CheckPolicyAuthSession(
762      COMMAND         *command,       // IN: primary parsing structure
763      UINT32           sessionIndex   // IN: index of session to be processed
764      )
765  {
766      SESSION             *session;
767      TPM2B_DIGEST         authPolicy;
768      TPMI_ALG_HASH        policyAlg;
769      UINT8                locality;
770  //
771      // Initialize pointer to the authorization session.
772      session = SessionGet(s_sessionHandles[sessionIndex]);
773
774      // If the command is TPM2_PolicySecret(), make sure that
775      // either password or authValue is required
776      if(command->code == TPM_CC_PolicySecret
777         &&  session->attributes.isPasswordNeeded == CLEAR
778         &&  session->attributes.isAuthValueNeeded == CLEAR)
779          return TPM_RC_MODE;
780      // See if the PCR counter for the session is still valid.
781      if(!SessionPCRValueIsCurrent(session))
782          return TPM_RC_PCR_CHANGED;
783      // Get authPolicy.
784      policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
785                                      &authPolicy);
786      // Compare authPolicy.
787      if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
788          return TPM_RC_POLICY_FAIL;
789      // Policy is OK so check if the other factors are correct
790
791      // Compare policy hash algorithm.
792      if(policyAlg != session->authHashAlg)
793          return TPM_RC_POLICY_FAIL;
794
795      // Compare timeout.
796      if(session->timeout != 0)
```

```
797         {
798             // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
799             // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
800             // a new nonce will be created just that, because TPM time can't advance
801             // we can't do time-based operations.
802             RETURN_IF_NV_IS_NOT_AVAILABLE;
803 
804             if((session->timeout < g_time)
805                || (session->epoch != g_timeEpoch))
806                 return TPM_RC_EXPIRED;
807         }
808         // If command code is provided it must match
809         if(session->commandCode != 0)
810         {
811             if(session->commandCode != command->code)
812                 return TPM_RC_POLICY_CC;
813         }
814         else
815         {
816             // If command requires a DUP or ADMIN authorization, the session must have
817             // command code set.
818             AUTH_ROLE   role = CommandAuthRole(command->index, sessionIndex);
819             if(role == AUTH_ADMIN || role == AUTH_DUP)
820                 return TPM_RC_POLICY_FAIL;
821         }
822         // Check command locality.
823         {
824             BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
825             BYTE        *buffer = sessionLocality;
826 
827             // Get existing locality setting in canonical form
828             sessionLocality[0] = 0;
829             TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
830 
831             // See if the locality has been set
832             if(sessionLocality[0] != 0)
833             {
834                 // If so, get the current locality
835                 locality = _plat__LocalityGet();
836                 if(locality < 5)
837                 {
838                     if(((sessionLocality[0] & (1 << locality)) == 0)
839                        || sessionLocality[0] > 31)
840                         return TPM_RC_LOCALITY;
841                 }
842                 else if(locality > 31)
843                 {
844                     if(sessionLocality[0] != locality)
845                         return TPM_RC_LOCALITY;
846                 }
847                 else
848                 {
849                     // Could throw an assert here but a locality error is just
850                     // as good. It just means that, whatever the locality is, it isn't
851                     // the locality requested so...
852                     return TPM_RC_LOCALITY;
853                 }
854             }
855         } // end of locality check
856         // Check physical presence.
857         if(session->attributes.isPPRequired == SET
858            && !_plat__PhysicalPresenceAsserted())
859             return TPM_RC_PP;
860         // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
861         // DUP role for this handle.
862         if(session->u1.cpHash.b.size != 0)
```

```
863         {
864             BOOL        OK;
865             if(session->attributes.isCpHashDefined)
866                 // Compare cpHash.
867                 OK = MemoryEqual2B(&session->u1.cpHash.b,
868                                    &ComputeCpHash(command, session->authHashAlg)->b);
869             else if(session->attributes.isTemplateSet)
870                 OK = CompareTemplateHash(command, session);
871             else
872                 OK = CompareNameHash(command, session);
873             if(!OK)
874                 return TPM_RCS_POLICY_FAIL;
875         }
876         if(session->attributes.checkNvWritten)
877         {
878             NV_REF           locator;
879             NV_INDEX        *nvIndex;
880     //
881             // If this is not an NV index, the policy makes no sense so fail it.
882             if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
883                 return TPM_RC_POLICY_FAIL;
884             // Get the index data
885             nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
886
887             // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
888             if((IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
889                != (session->attributes.nvWrittenState == SET))
890                 return TPM_RC_POLICY_FAIL;
891         }
892         return TPM_RC_SUCCESS;
893     }
```

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

```
894     static TPM_RC
895     RetrieveSessionData(
896         COMMAND         *command        // IN: main parsing structure for command
897         )
898     {
899         int              i;
900         TPM_RC           result;
901         SESSION         *session;
902         TPMA_SESSION     sessionAttributes;
903         TPM_HT           sessionType;
904         INT32            sessionIndex;
905         TPM_RC           errorIndex;
906     //
907         s_decryptSessionIndex = UNDEFINED_INDEX;
908         s_encryptSessionIndex = UNDEFINED_INDEX;
909         s_auditSessionIndex = UNDEFINED_INDEX;
910
911         for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
912         {
913             errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
```

```
914
915         // If maximum allowed number of sessions has been parsed, return a size
916         // error with a session number that is larger than the number of allowed
917         // sessions
918         if(sessionIndex == MAX_SESSION_NUM)
919             return TPM_RCS_SIZE + errorIndex;
920         // make sure that the associated handle for each session starts out
921         // unassigned
922         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
923
924         // First parameter: Session handle.
925         result = TPMI_SH_AUTH_SESSION_Unmarshal(
926             &s_sessionHandles[sessionIndex],
927             &command->parameterBuffer,
928             &command->authSize, TRUE);
929         if(result != TPM_RC_SUCCESS)
930             return result + TPM_RC_S + g_rcIndex[sessionIndex];
931         // Second parameter: Nonce.
932         result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
933                                        &command->parameterBuffer,
934                                        &command->authSize);
935         if(result != TPM_RC_SUCCESS)
936             return result + TPM_RC_S + g_rcIndex[sessionIndex];
937         // Third parameter: sessionAttributes.
938         result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
939                                         &command->parameterBuffer,
940                                         &command->authSize);
941         if(result != TPM_RC_SUCCESS)
942             return result + TPM_RC_S + g_rcIndex[sessionIndex];
943         // Fourth parameter: authValue (PW or HMAC).
944         result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
945                                       &command->parameterBuffer,
946                                       &command->authSize);
947         if(result != TPM_RC_SUCCESS)
948             return result + errorIndex;
949
950         sessionAttributes = s_attributes[sessionIndex];
951         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
952         {
953             // A PWAP session needs additional processing.
954             //     Can't have any attributes set other than continueSession bit
955             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
956                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
957                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
958                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
959                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
960                 return TPM_RCS_ATTRIBUTES + errorIndex;
961             //     The nonce size must be zero.
962             if(s_nonceCaller[sessionIndex].t.size != 0)
963                 return TPM_RCS_NONCE + errorIndex;
964             continue;
965         }
966         // For not password sessions...
967         // Find out if the session is loaded.
968         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
969             return TPM_RC_REFERENCE_S0 + sessionIndex;
970         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
971         session = SessionGet(s_sessionHandles[sessionIndex]);
972
973         // Check if the session is an HMAC/policy session.
974         if((session->attributes.isPolicy == SET
975             && sessionType == TPM_HT_HMAC_SESSION)
976            || (session->attributes.isPolicy == CLEAR
977                && sessionType == TPM_HT_POLICY_SESSION))
978             return TPM_RCS_HANDLE + errorIndex;
979         // Check that this handle has not previously been used.
```

```
980             for(i = 0; i < sessionIndex; i++)
981             {
982                 if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
983                     return TPM_RCS_HANDLE + errorIndex;
984             }
985             // If the session is used for parameter encryption or audit as well, set
986             // the corresponding Indexes.
987
988             // First process decrypt.
989             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
990             {
991                 // Check if the commandCode allows command parameter encryption.
992                 if(DecryptSize(command->index) == 0)
993                     return TPM_RCS_ATTRIBUTES + errorIndex;
994                 // Encrypt attribute can only appear in one session
995                 if(s_decryptSessionIndex != UNDEFINED_INDEX)
996                     return TPM_RCS_ATTRIBUTES + errorIndex;
997                 // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
998                 if(session->symmetric.algorithm == TPM_ALG_NULL)
999                     return TPM_RCS_SYMMETRIC + errorIndex;
1000                // All checks passed, so set the index for the session used to decrypt
1001                // a command parameter.
1002                s_decryptSessionIndex = sessionIndex;
1003            }
1004            // Now process encrypt.
1005            if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))
1006            {
1007                // Check if the commandCode allows response parameter encryption.
1008                if(EncryptSize(command->index) == 0)
1009                    return TPM_RCS_ATTRIBUTES + errorIndex;
1010                // Encrypt attribute can only appear in one session.
1011                if(s_encryptSessionIndex != UNDEFINED_INDEX)
1012                    return TPM_RCS_ATTRIBUTES + errorIndex;
1013                // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
1014                if(session->symmetric.algorithm == TPM_ALG_NULL)
1015                    return TPM_RCS_SYMMETRIC + errorIndex;
1016                // All checks passed, so set the index for the session used to encrypt
1017                // a response parameter.
1018                s_encryptSessionIndex = sessionIndex;
1019            }
1020            // At last process audit.
1021            if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))
1022            {
1023                // Audit attribute can only appear in one session.
1024                if(s_auditSessionIndex != UNDEFINED_INDEX)
1025                    return TPM_RCS_ATTRIBUTES + errorIndex;
1026                // An audit session can not be policy session.
1027                if(HandleGetType(s_sessionHandles[sessionIndex])
1028                   == TPM_HT_POLICY_SESSION)
1029                    return TPM_RCS_ATTRIBUTES + errorIndex;
1030                // If this is a reset of the audit session, or the first use
1031                // of the session as an audit session, it doesn't matter what
1032                // the exclusive state is. The session will become exclusive.
1033                if(!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)
1034                   && session->attributes.isAudit == SET)
1035                {
1036                    // Not first use or reset. If auditExlusive is SET, then this
1037                    // session must be the current exclusive session.
1038                    if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1039                       && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
1040                        return TPM_RC_EXCLUSIVE;
1041                }
1042                s_auditSessionIndex = sessionIndex;
1043            }
1044            // Initialize associated handle as undefined. This will be changed when
1045            // the handles are processed.
```

```
1046            s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1047        }
1048        command->sessionNum = sessionIndex;
1049        return TPM_RC_SUCCESS;
1050    }
```

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

```
1051    static TPM_RC
1052    CheckLockedOut(
1053        BOOL             lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
1054        )
1055    {
1056        // If NV is unavailable, and current cycle state recorded in NV is not
1057        // SU_NONE_VALUE, refuse to check any authorization because we would
1058        // not be able to handle a DA failure.
1059        if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
1060            return g_NvStatus;
1061        // Check if DA info needs to be updated in NV.
1062        if(s_DAPendingOnNV)
1063        {
1064            // If NV is accessible,
1065            RETURN_IF_NV_IS_NOT_AVAILABLE;
1066
1067            // ... write the pending DA data and proceed.
1068            NV_SYNC_PERSISTENT(lockOutAuthEnabled);
1069            NV_SYNC_PERSISTENT(failedTries);
1070            s_DAPendingOnNV = FALSE;
1071        }
1072        // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
1073        // is disabled...
1074        if(lockoutAuthCheck)
1075        {
1076            if(gp.lockOutAuthEnabled == FALSE)
1077                return TPM_RC_LOCKOUT;
1078        }
1079        else
1080        {
1081            // ... or if the number of failed tries has been maxed out.
1082            if(gp.failedTries >= gp.maxTries)
1083                return TPM_RC_LOCKOUT;
1084    #if USE_DA_USED
1085            // If the daUsed flag is not SET, then no DA validation until the
1086            // daUsed state is written to NV
1087            if(!g_daUsed)
1088            {
1089                RETURN_IF_NV_IS_NOT_AVAILABLE;
1090                g_daUsed = TRUE;
1091                gp.orderlyState = SU_DA_USED_VALUE;
1092                NV_SYNC_PERSISTENT(orderlyState);
1093                return TPM_RC_RETRY;
```

```
1094            }
1095    #endif
1096        }
1097        return TPM_RC_SUCCESS;
1098    }
```

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

```
1099    static TPM_RC
1100    CheckAuthSession(
1101        COMMAND         *command,       // IN: primary parsing structure
1102        UINT32           sessionIndex   // IN: index of session to be processed
1103        )
1104    {
1105        TPM_RC           result = TPM_RC_SUCCESS;
1106        SESSION         *session = NULL;
1107        TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
1108        TPM_HANDLE       associatedHandle = s_associatedHandles[sessionIndex];
1109        TPM_HT           sessionHandleType = HandleGetType(sessionHandle);
1110        BOOL             authUsed;
1111    //
1112        pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1113
1114        // Take care of physical presence
1115        if(associatedHandle == TPM_RH_PLATFORM)
1116        {
1117            // If the physical presence is required for this command, check for PP
1118            // assertion. If it isn't asserted, no point going any further.
1119            if(PhysicalPresenceIsRequired(command->index)
1120               && !_plat__PhysicalPresenceAsserted())
1121                return TPM_RC_PP;
1122        }
1123        if(sessionHandle != TPM_RS_PW)
1124        {
1125            session = SessionGet(sessionHandle);
1126
1127            // Set includeAuth to indicate if DA checking will be required and if the
1128            // authValue will be included in any HMAC.
1129            if(sessionHandleType == TPM_HT_POLICY_SESSION)
1130            {
1131                // For a policy session, will check the DA status of the entity if either
```

```
1132                // isAuthValueNeeded or isPasswordNeeded is SET.
1133                session->attributes.includeAuth =
1134                    session->attributes.isAuthValueNeeded
1135                    || session->attributes.isPasswordNeeded;
1136            }
1137            else
1138            {
1139                // For an HMAC session, need to check unless the session
1140                // is bound.
1141                session->attributes.includeAuth =
1142                    !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1143            }
1144            authUsed = session->attributes.includeAuth;
1145        }
1146        else
1147            // Password session
1148            authUsed = TRUE;
1149        // If the authorization session is going to use an authValue, then make sure
1150        // that access to that authValue isn't locked out.
1151        if(authUsed)
1152        {
1153            // See if entity is subject to lockout.
1154            if(!IsDAExempted(associatedHandle))
1155            {
1156                // See if in lockout
1157                result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1158                if(result != TPM_RC_SUCCESS)
1159                    return result;
1160            }
1161        }
1162        // Policy or HMAC+PW?
1163        if(sessionHandleType != TPM_HT_POLICY_SESSION)
1164        {
1165            // for non-policy session make sure that a policy session is not required
1166            if(IsPolicySessionRequired(command->index, sessionIndex))
1167                return TPM_RC_AUTH_TYPE;
1168            // The authValue must be available.
1169            // Note: The authValue is going to be "used" even if it is an EmptyAuth.
1170            // and the session is bound.
1171            if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1172                return TPM_RC_AUTH_UNAVAILABLE;
1173        }
1174        else
1175        {
1176            // ... see if the entity has a policy, ...
1177            // Note: IsAuthPolicyAvalable will return FALSE if the sensitive area of the
1178            // object is not loaded
1179            if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1180                return TPM_RC_AUTH_UNAVAILABLE;
1181            // ... and check the policy session.
1182            result = CheckPolicyAuthSession(command, sessionIndex);
1183            if(result != TPM_RC_SUCCESS)
1184                return result;
1185        }
1186        // Check authorization according to the type
1187        if((TPM_RS_PW == sessionHandle) || (session->attributes.isPasswordNeeded == SET))
1188            result = CheckPWAuthSession(sessionIndex);
1189        else
1190            result = CheckSessionHMAC(command, sessionIndex);
1191        // Do processing for PIN Indexes are only three possibilities for 'result' at
1192        // this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
1193        // For all these cases, we would have to process a PIN index if the
1194        // authValue of the index was used for authorization.
1195        if((TPM_HT_NV_INDEX == HandleGetType(associatedHandle)) && authUsed)
1196        {
1197            NV_REF           locator;
```

```
1198            NV_INDEX        *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1199            NV_PIN           pinData;
1200            TPMA_NV          nvAttributes;
1201    //
1202            pAssert(nvIndex != NULL);
1203            nvAttributes = nvIndex->publicArea.attributes;
1204            // If this is a PIN FAIL index and the value has been written
1205            // then we can update the counter (increment or clear)
1206            if(IsNvPinFailIndex(nvAttributes)
1207               && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
1208            {
1209                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1210                if(result != TPM_RC_SUCCESS)
1211                    pinData.pin.pinCount++;
1212                else
1213                    pinData.pin.pinCount = 0;
1214                NvWriteUINT64Data(nvIndex, pinData.intVal);
1215            }
1216            // If this is a PIN PASS Index, increment if we have used the
1217            // authorization value.
1218            // NOTE: If the counter has already hit the limit, then we
1219            // would not get here because the authorization value would not
1220            // be available and the TPM would have returned before it gets here
1221            else if(IsNvPinPassIndex(nvAttributes)
1222                    && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)
1223                    && result == TPM_RC_SUCCESS)
1224            {
1225                // If the access is valid, then increment the use counter
1226                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1227                pinData.pin.pinCount++;
1228                NvWriteUINT64Data(nvIndex, pinData.intVal);
1229            }
1230        }
1231        return result;
1232    }
1233    #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

```
1234    static TPM_RC
1235    CheckCommandAudit(
1236        COMMAND         *command
1237        )
1238    {
1239        // If the audit digest is clear and command audit is required, NV must be
1240        // available so that TPM2_GetCommandAuditDigest() is able to increment
1241        // audit counter. If NV is not available, the function bails out to prevent
1242        // the TPM from attempting an operation that would fail anyway.
1243        if(gr.commandAuditDigest.t.size == 0
1244           || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1245        {
1246            RETURN_IF_NV_IS_NOT_AVAILABLE;
1247        }
1248        // Make sure that the cpHash is computed for the algorithm
1249        ComputeCpHash(command, gp.auditHashAlg);
```

```
1250        return TPM_RC_SUCCESS;
1251    }
1252    #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

```
1253    TPM_RC
1254    ParseSessionBuffer(
1255        COMMAND         *command        // IN: the structure that contains
1256        )
1257    {
1258        TPM_RC              result;
1259        UINT32              i;
1260        INT32               size = 0;
1261        TPM2B_AUTH          extraKey;
1262        UINT32              sessionIndex;
1263        TPM_RC              errorIndex;
1264        SESSION            *session = NULL;
1265    //
1266        // Check if a command allows any session in its session area.
1267        if(!IsSessionAllowed(command->index))
1268            return TPM_RC_AUTH_CONTEXT;
1269        // Default-initialization.
1270        command->sessionNum = 0;
1271
1272        result = RetrieveSessionData(command);
1273        if(result != TPM_RC_SUCCESS)
1274            return result;
1275        // There is no command in the TPM spec that has more handles than
1276        // MAX_SESSION_NUM.
1277        pAssert(command->handleNum <= MAX_SESSION_NUM);
1278
1279        // Associate the session with an authorization handle.
1280        for(i = 0; i < command->handleNum; i++)
1281        {
1282            if(CommandAuthRole(command->index, i) != AUTH_NONE)
1283            {
1284                // If the received session number is less than the number of handles
1285                // that requires authorization, an error should be returned.
1286                // Note: for all the TPM 2.0 commands, handles requiring
1287                // authorization come first in a command input and there are only ever
1288                // two values requiring authorization
1289                if(i > (command->sessionNum - 1))
1290                    return TPM_RC_AUTH_MISSING;
1291                // Record the handle associated with the authorization session
1292                s_associatedHandles[i] = command->handles[i];
1293            }
1294        }
1295        // Consistency checks are done first to avoid authorization failure when the
1296        // command will not be executed anyway.
1297        for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1298        {
1299            errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1300            // PW session must be an authorization session
1301            if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1302            {
```

```
1303                if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1304                    return TPM_RCS_HANDLE + errorIndex;
1305                // a password session can't be audit, encrypt or decrypt
1306                if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1307                   || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1308                   || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1309                    return TPM_RCS_ATTRIBUTES + errorIndex;
1310                session = NULL;
1311            }
1312            else
1313            {
1314                session = SessionGet(s_sessionHandles[sessionIndex]);
1315
1316                // A trial session can not appear in session area, because it cannot
1317                // be used for authorization, audit or encrypt/decrypt.
1318                if(session->attributes.isTrialPolicy == SET)
1319                    return TPM_RCS_ATTRIBUTES + errorIndex;
1320
1321                // See if the session is bound to a DA protected entity
1322                // NOTE: Since a policy session is never bound, a policy is still
1323                // usable even if the object is DA protected and the TPM is in
1324                // lockout.
1325                if(session->attributes.isDaBound == SET)
1326                {
1327                    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1328                    if(result != TPM_RC_SUCCESS)
1329                        return result;
1330                }
1331                // If this session is for auditing, make sure the cpHash is computed.
1332                if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
1333                    ComputeCpHash(command, session->authHashAlg);
1334            }
1335
1336            // if the session has an associated handle, check the authorization
1337            if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1338            {
1339                result = CheckAuthSession(command, sessionIndex);
1340                if(result != TPM_RC_SUCCESS)
1341                    return RcSafeAddToResult(result, errorIndex);
1342            }
1343            else
1344            {
1345                // a session that is not for authorization must either be encrypt,
1346                // decrypt, or audit
1347                if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1348                   &&  !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1349                   &&  !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1350                    return TPM_RCS_ATTRIBUTES + errorIndex;
1351
1352                // no authValue included in any of the HMAC computations
1353                pAssert(session != NULL);
1354                session->attributes.includeAuth = CLEAR;
1355
1356                // check HMAC for encrypt/decrypt/audit only sessions
1357                result = CheckSessionHMAC(command, sessionIndex);
1358                if(result != TPM_RC_SUCCESS)
1359                    return RcSafeAddToResult(result, errorIndex);
1360            }
1361        }
1362    #ifdef  TPM_CC_GetCommandAuditDigest
1363        // Check if the command should be audited. Need to do this before any parameter
1364        // encryption so that the cpHash for the audit is correct
1365        if(CommandAuditIsRequired(command->index))
1366        {
1367            result = CheckCommandAudit(command);
1368            if(result != TPM_RC_SUCCESS)
```

```
1369                return result;              // No session number to reference
1370        }
1371    #endif
1372        // Decrypt the first parameter if applicable. This should be the last operation
1373        // in session processing.
1374        // If the encrypt session is associated with a handle and the handle's
1375        // authValue is available, then authValue is concatenated with sessionKey to
1376        // generate encryption key, no matter if the handle is the session bound entity
1377        // or not.
1378        if(s_decryptSessionIndex != UNDEFINED_INDEX)
1379        {
1380            // If this is an authorization session, include the authValue in the
1381            // generation of the decryption key
1382            if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1383            {
1384                EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1385                                   &extraKey);
1386            }
1387            else
1388            {
1389                extraKey.b.size = 0;
1390            }
1391            size = DecryptSize(command->index);
1392            result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1393                                              &s_nonceCaller[s_decryptSessionIndex].b,
1394                                              command->parameterSize, (UINT16)size,
1395                                              &extraKey,
1396                                              command->parameterBuffer);
1397            if(result != TPM_RC_SUCCESS)
1398                return RcSafeAddToResult(result,
1399                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1400        }
1401
1402        return TPM_RC_SUCCESS;
1403    }
```

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

```
1404    TPM_RC
1405    CheckAuthNoSession(
1406        COMMAND         *command        // IN: command parsing structure
1407        )
1408    {
1409        UINT32 i;
1410        TPM_RC           result = TPM_RC_SUCCESS;
1411    //
1412        // Check if the command requires authorization
1413        for(i = 0; i < command->handleNum; i++)
1414        {
1415            if(CommandAuthRole(command->index, i) != AUTH_NONE)
1416                return TPM_RC_AUTH_MISSING;
1417        }
1418    #ifdef  TPM_CC_GetCommandAuditDigest
1419        // Check if the command should be audited.
1420        if(CommandAuditIsRequired(command->index))
1421        {
1422            result = CheckCommandAudit(command);
```

```
1423            if(result != TPM_RC_SUCCESS)
1424                return result;
1425        }
1426    #endif
1427        // Initialize number of sessions to be 0
1428        command->sessionNum = 0;
1429
1430        return TPM_RC_SUCCESS;
1431    }
```

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.

```
1432    static TPM2B_DIGEST *
1433    ComputeRpHash(
1434        COMMAND         *command,       // IN: command structure
1435        TPM_ALG_ID       hashAlg        // IN: hash algorithm to compute rpHash
1436        )
1437    {
1438        TPM2B_DIGEST    *rpHash = GetRpHashPointer(command, hashAlg);
1439        HASH_STATE       hashState;
1440    //
1441        if(rpHash->t.size == 0)
1442        {
1443        //   rpHash := hash(responseCode || commandCode || parameters)
1444
1445        // Initiate hash creation.
1446            rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1447
1448            // Add hash constituents.
1449            CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1450            CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1451            CryptDigestUpdate(&hashState, command->parameterSize,
1452                              command->parameterBuffer);
1453            // Complete hash computation.
1454            CryptHashEnd2B(&hashState, &rpHash->b);
1455        }
1456        return rpHash;
1457    }
```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1458    static void
1459    InitAuditSession(
1460        SESSION         *session        // session to be initialized
1461        )
1462    {
1463        // Mark session as an audit session.
1464        session->attributes.isAudit = SET;
1465
```

```
1466        // Audit session can not be bound.
1467        session->attributes.isBound = CLEAR;
1468
1469        // Size of the audit log is the size of session hash algorithm digest.
1470        session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1471
1472        // Set the original digest value to be 0.
1473        MemorySet(&session->u2.auditDigest.t.buffer,
1474                  0,
1475                  session->u2.auditDigest.t.size);
1476        return;
1477    }
```

6.4.5.4 UpdateAuditDigest

Function to update an audit digest

```
1478    static void
1479    UpdateAuditDigest(
1480        COMMAND         *command,
1481        TPMI_ALG_HASH    hashAlg,
1482        TPM2B_DIGEST    *digest
1483        )
1484    {
1485        HASH_STATE       hashState;
1486        TPM2B_DIGEST    *cpHash = GetCpHash(command, hashAlg);
1487        TPM2B_DIGEST    *rpHash = ComputeRpHash(command, hashAlg);
1488    //
1489        pAssert(cpHash != NULL);
1490
1491        // digestNew :=  hash (digestOld || cpHash || rpHash)
1492            // Start hash computation.
1493        digest->t.size = CryptHashStart(&hashState, hashAlg);
1494            // Add old digest.
1495        CryptDigestUpdate2B(&hashState, &digest->b);
1496            // Add cpHash
1497        CryptDigestUpdate2B(&hashState, &cpHash->b);
1498            // Add rpHash
1499        CryptDigestUpdate2B(&hashState, &rpHash->b);
1500            // Finalize the hash.
1501        CryptHashEnd2B(&hashState, &digest->b);
1502    }
```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```
1503    static void
1504    Audit(
1505        COMMAND         *command,       // IN: primary control structure
1506        SESSION         *auditSession   // IN: loaded audit session
1507        )
1508    {
1509        UpdateAuditDigest(command, auditSession->authHashAlg,
1510                          &auditSession->u2.auditDigest);
1511        return;
1512    }
1513    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

```
1514    static void
1515    CommandAudit(
1516        COMMAND         *command        // IN:
1517        )
1518    {
1519        // If the digest.size is one, it indicates the special case of changing
1520        // the audit hash algorithm. For this case, no audit is done on exit.
1521        // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1522        // force an update to the NV on exit so that the change in digest will
1523        // be recorded. So, it is safe to exit here without setting any flags
1524        // because the digest change will be written to NV when this code exits.
1525        if(gr.commandAuditDigest.t.size == 1)
1526        {
1527            gr.commandAuditDigest.t.size = 0;
1528            return;
1529        }
1530        // If the digest size is zero, need to start a new digest and increment
1531        // the audit counter.
1532        if(gr.commandAuditDigest.t.size == 0)
1533        {
1534            gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1535            MemorySet(gr.commandAuditDigest.t.buffer,
1536                      0,
1537                      gr.commandAuditDigest.t.size);
1538
1539            // Bump the counter and save its value to NV.
1540            gp.auditCounter++;
1541            NV_SYNC_PERSISTENT(auditCounter);
1542        }
1543        UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1544        return;
1545    }
1546    #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.

```
1547    static void
1548    UpdateAuditSessionStatus(
1549        COMMAND         *command        // IN: primary control structure
1550        )
1551    {
1552        UINT32           i;
1553        TPM_HANDLE       auditSession = TPM_RH_UNASSIGNED;
1554    //
1555        // Iterate through sessions
1556        for(i = 0; i < command->sessionNum; i++)
1557        {
1558            SESSION     *session;
1559    //
1560            // PW session do not have a loaded session and can not be an audit
1561            // session either.  Skip it.
1562            if(s_sessionHandles[i] == TPM_RS_PW)
1563                continue;
1564            session = SessionGet(s_sessionHandles[i]);
```

```
1565
1566            // If a session is used for audit
1567            if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1568            {
1569                // An audit session has been found
1570                auditSession = s_sessionHandles[i];
1571
1572                // If the session has not been an audit session yet, or
1573                // the auditSetting bits indicate a reset, initialize it and set
1574                // it to be the exclusive session
1575                if(session->attributes.isAudit == CLEAR
1576                   || IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditReset))
1577                {
1578                    InitAuditSession(session);
1579                    g_exclusiveAuditSession = auditSession;
1580                }
1581                else
1582                {
1583                    // Check if the audit session is the current exclusive audit
1584                    // session and, if not, clear previous exclusive audit session.
1585                    if(g_exclusiveAuditSession != auditSession)
1586                        g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1587                }
1588                // Report audit session exclusivity.
1589                if(g_exclusiveAuditSession == auditSession)
1590                {
1591                    SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1592                }
1593                else
1594                {
1595                    CLEAR_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1596                }
1597                // Extend audit log.
1598                Audit(command, session);
1599            }
1600        }
1601        // If no audit session is found in the command, and the command allows
1602        // a session then, clear the current exclusive
1603        // audit session.
1604        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1605        {
1606            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1607        }
1608        return;
1609    }
```

6.4.5.8 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1610    static void
1611    ComputeResponseHMAC(
1612        COMMAND         *command,       // IN: command structure
1613        UINT32           sessionIndex,  // IN: session index to be processed
1614        SESSION         *session,       // IN: loaded session
1615        TPM2B_DIGEST    *hmac           // OUT: authHMAC
1616        )
1617    {
1618        TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1619        TPM2B_KEY        key;       // HMAC key
1620        BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1621        BYTE            *buffer;
1622        UINT32           marshalSize;
1623        HMAC_STATE       hmacState;
```

```
1624        TPM2B_DIGEST    *rpHash = ComputeRpHash(command, session->authHashAlg);
1625    //
1626        // Generate HMAC key
1627        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1628
1629        // Add the object authValue if required
1630        if(session->attributes.includeAuth == SET)
1631        {
1632            // Note: includeAuth may be SET for a policy that is used in
1633            // UndefineSpaceSpecial(). At this point, the Index has been deleted
1634            // so the includeAuth will have no meaning. However, the
1635            // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1636            // this will return the authValue associated with TPM_RH_NULL and that is
1637            // and empty buffer.
1638            TPM2B_AUTH          authValue;
1639    //
1640            // Get the authValue with trailing zeros removed
1641            EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1642
1643            // Add it to the key
1644            MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1645        }
1646
1647        // if the HMAC key size is 0, the response HMAC is computed according to the
1648        // input HMAC
1649        if(key.t.size == 0
1650           && s_inputAuthValues[sessionIndex].t.size == 0)
1651        {
1652            hmac->t.size = 0;
1653            return;
1654        }
1655        // Start HMAC computation.
1656        hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1657
1658        // Add hash components.
1659        CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
1660        CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1661        CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1662
1663        // Add session attributes.
1664        buffer = marshalBuffer;
1665        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1666        CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1667
1668        // Finalize HMAC.
1669        CryptHmacEnd2B(&hmacState, &hmac->b);
1670
1671        return;
1672    }
```

6.4.5.9 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
1673    static void
1674    UpdateInternalSession(
1675        SESSION         *session,       // IN: the session structure
1676        UINT32           i              // IN: session number
1677        )
1678    {
1679        // If nonce is rolling in a policy session, the policy related data
```

```
1680        // will be re-initialized.
1681        if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1682           && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1683        {
1684            // When the nonce rolls it starts a new timing interval for the
1685            // policy session.
1686            SessionResetPolicyData(session);
1687            SessionSetStartTime(session);
1688        }
1689        return;
1690    }
```

6.4.5.10 BuildSingleResponseAuth()

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

```
1691    static TPM2B_NONCE *
1692    BuildSingleResponseAuth(
1693        COMMAND         *command,       // IN: command structure
1694        UINT32           sessionIndex,  // IN: session index to be processed
1695        TPM2B_AUTH      *auth           // OUT: authHMAC
1696        )
1697    {
1698        // Fill in policy/HMAC based session response.
1699        SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);
1700    //
1701        // If the session is a policy session with isPasswordNeeded SET, the
1702        // authorization field is empty.
1703        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1704           && session->attributes.isPasswordNeeded == SET)
1705            auth->t.size = 0;
1706        else
1707            // Compute response HMAC.
1708            ComputeResponseHMAC(command, sessionIndex, session, auth);
1709
1710        UpdateInternalSession(session, sessionIndex);
1711        return &session->nonceTPM;
1712    }
```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```
1713    static void
1714    UpdateAllNonceTPM(
1715        COMMAND         *command        // IN: controlling structure
1716        )
1717    {
1718        UINT32      i;
1719        SESSION     *session;
1720    //
1721        for(i = 0; i < command->sessionNum; i++)
1722        {
1723            // If not a PW session, compute the new nonceTPM.
1724            if(s_sessionHandles[i] != TPM_RS_PW)
1725            {
1726                session = SessionGet(s_sessionHandles[i]);
1727                // Update nonceTPM in both internal session and response.
1728                CryptRandomGenerate(session->nonceTPM.t.size,
1729                                    session->nonceTPM.t.buffer);
1730            }
1731        }
1732        return;
```

```
1733    }
```

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.

```
1734    void
1735    BuildResponseSession(
1736        COMMAND         *command        // IN: structure that has relevant command
1737                                        //     information
1738        )
1739    {
1740        pAssert(command->authSize == 0);
1741
1742        // Reset the parameter buffer to point to the start of the parameters so that
1743        // there is a starting point for any rpHash that might be generated and so there
1744        // is a place where parameter encryption would start
1745        command->parameterBuffer = command->responseBuffer - command->parameterSize;
1746
1747        // Session nonces should be updated before parameter encryption
1748        if(command->tag == TPM_ST_SESSIONS)
1749        {
1750            UpdateAllNonceTPM(command);
1751
1752            // Encrypt first parameter if applicable. Parameter encryption should
1753            // happen after nonce update and before any rpHash is computed.
1754            // If the encrypt session is associated with a handle, the authValue of
1755            // this handle will be concatenated with sessionKey to generate
1756            // encryption key, no matter if the handle is the session bound entity
1757            // or not. The authValue is added to sessionKey only when the authValue
1758            // is available.
1759            if(s_encryptSessionIndex != UNDEFINED_INDEX)
1760            {
1761                UINT32          size;
1762                TPM2B_AUTH      extraKey;
1763    //
1764                extraKey.b.size = 0;
1765                // If this is an authorization session, include the authValue in the
1766                // generation of the encryption key
1767                if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
1768                {
1769                    EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1770                                       &extraKey);
1771                }
1772                size = EncryptSize(command->index);
1773                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1774                                         &s_nonceCaller[s_encryptSessionIndex].b,
1775                                         (UINT16)size,
1776                                         &extraKey,
1777                                         command->parameterBuffer);
1778            }
1779        }
1780        // Audit sessions should be processed regardless of the tag because
1781        // a command with no session may cause a change of the exclusivity state.
1782        UpdateAuditSessionStatus(command);
1783    #if CC_GetCommandAuditDigest
1784        // Command Audit
1785        if(CommandAuditIsRequired(command->index))
1786            CommandAudit(command);
1787    #endif
```

```
1788        // Process command with sessions.
1789        if(command->tag == TPM_ST_SESSIONS)
1790        {
1791            UINT32           i;
1792    //
1793            pAssert(command->sessionNum > 0);
1794
1795            // Iterate over each session in the command session area, and create
1796            // corresponding sessions for response.
1797            for(i = 0; i < command->sessionNum; i++)
1798            {
1799                TPM2B_NONCE     *nonceTPM;
1800                TPM2B_DIGEST     responseAuth;
1801                // Make sure that continueSession is SET on any Password session.
1802                // This makes it marginally easier for the management software
1803                // to keep track of the closed sessions.
1804                if(s_sessionHandles[i] == TPM_RS_PW)
1805                {
1806                    SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession);
1807                    responseAuth.t.size = 0;
1808                    nonceTPM = (TPM2B_NONCE *)&responseAuth;
1809                }
1810                else
1811                {
1812                    // Compute the response HMAC and get a pointer to the nonce used.
1813                    // This function will also update the values if needed. Note, the
1814                    nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
1815                }
1816                command->authSize += TPM2B_NONCE_Marshal(nonceTPM,
1817                                                         &command->responseBuffer,
1818                                                         NULL);
1819                command->authSize += TPMA_SESSION_Marshal(&s_attributes[i],
1820                                                          &command->responseBuffer,
1821                                                          NULL);
1822                command->authSize += TPM2B_DIGEST_Marshal(&responseAuth,
1823                                                          &command->responseBuffer,
1824                                                          NULL);
1825                if(!IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1826                    SessionFlush(s_sessionHandles[i]);
1827            }
1828        }
1829        return;
1830    }
```

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().

```
1831    void
1832    SessionRemoveAssociationToHandle(
1833        TPM_HANDLE       handle
1834        )
1835    {
1836        UINT32               i;
1837    //
1838        for(i = 0; i < MAX_SESSION_NUM; i++)
1839        {
1840            if(s_associatedHandles[i] == handle)
1841            {
1842                s_associatedHandles[i] = TPM_RH_NULL;
1843            }
1844        }
1845    }
```

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

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

```
TPM_RC
PolicyParameterChecks(
    SESSION         *session,
    UINT64           authTimeout,
    TPM2B_DIGEST    *cpHashA,
    TPM2B_NONCE     *nonce,
    TPM_RC           blameNonce,
    TPM_RC           blameCpHash,
    TPM_RC           blameExpiration
    )
{
    // Validate that input nonceTPM is correct if present
    if(nonce != NULL && nonce->t.size != 0)
    {
        if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
            return TPM_RCS_NONCE + blameNonce;
    }
    // If authTimeout is set (expiration != 0...
    if(authTimeout != 0)
    {
        // Validate input expiration.
        // Cannot compare time if clock stop advancing.  A TPM_RC_NV_UNAVAILABLE
        // or TPM_RC_NV_RATE error may be returned here.
        RETURN_IF_NV_IS_NOT_AVAILABLE;

        // if the time has already passed or the time epoch has changed then the
        // time value is no longer good.
        if((authTimeout < g_time)
           || (session->epoch != g_timeEpoch))
            return TPM_RCS_EXPIRED + blameExpiration;
    }
    // If the cpHash is present, then check it
    if(cpHashA != NULL && cpHashA->t.size != 0)
    {
        // The cpHash input has to have the correct size
        if(cpHashA->t.size != session->u2.policyDigest.t.size)
            return TPM_RCS_SIZE + blameCpHash;

        // If the cpHash has already been set, then this input value
        // must match the current value.
        if(session->u1.cpHash.b.size != 0
           && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
            return TPM_RC_CPHASH;
    }
```

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

```
TPM_RC
NvReadAccessChecks(
    TPM_HANDLE          authHandle,     // IN: the handle that provided the
                                        //     authorization
    TPM_HANDLE          nvHandle,       // IN: the handle of the NV index to be read
    TPMA_NV             attributes      // IN: the attributes of 'nvHandle'
    )
{
    // If data is read locked, returns an error
   if(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

```
86  BOOL
87  NvIsPinPassIndex(
88      TPM_HANDLE          index       // IN: Handle to check
89      )
90  {
91      if(HandleGetType(index) == TPM_HT_NV_INDEX)
92      {
93          NV_INDEX                *nvIndex = NvGetIndexInfo(index, NULL);
94
95          return IsNvPinPassIndex(nvIndex->publicArea.attributes);
96      }
97      return FALSE;
98  }
```

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
    )
```

```
{
    const TPM2B         *seed = seedIn;

    // Determine the algorithms for the KDF and the encryption/decryption
    // For TPM_RH_NULL, using context settings
    if(protector == NULL)
    {
        // Use the context encryption algorithm and key size
        *symAlg = CONTEXT_ENCRYPT_ALG;
        symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
        *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    }
    else
    {
        TPMT_SYM_DEF_OBJECT *symDef;
        symDef = &protector->publicArea.parameters.asymDetail.symmetric;
        *symAlg = symDef->algorithm;
        *keyBits = symDef->keyBits.sym;
        symKey->t.size = (*keyBits + 7) / 8;
    }
    // Get seed for KDF
    if(seed == NULL)
        seed = GetSeedForKDF(protector);
    // KDFa to generate symmetric key and IV value
    CryptKDFa(hashAlg, seed, STORAGE_KEY, name, NULL,
              symKey->t.size * 8, symKey->t.buffer, NULL, FALSE);
    return;
}
```

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.

```
static void
ComputeOuterIntegrity(
    TPM2B           *name,              // IN: the name of the object
    OBJECT          *protector,         // IN: the object that
                                        //     provides protection. For an object,
                                        //     it is a parent. For a credential, it
                                        //     is the encrypt object. For
                                        //     a Temporary Object, it is NULL
    TPMI_ALG_HASH    hashAlg,           // IN: algorithm to use for integrity
    TPM2B           *seedIn,            // IN: an external seed may be provided for
                                        //     duplication blob. For non duplication
                                        //     blob, this parameter should be NULL
    UINT32           sensitiveSize,     // IN: size of the marshaled sensitive data
    BYTE            *sensitiveData,     // IN: sensitive area
    TPM2B_DIGEST    *integrity          // OUT: integrity
    )
{
    HMAC_STATE       hmacState;
    TPM2B_DIGEST     hmacKey;
    const TPM2B     *seed = seedIn;
//
    // Get seed for KDF
    if(seed == NULL)
        seed = GetSeedForKDF(protector);
    // Determine the HMAC key bits
    hmacKey.t.size = CryptHashGetDigestSize(hashAlg);

    // 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
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 TPM_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                  // TPM_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             // TPM_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 = TPMA_ZERO_INITIALIZER();
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 = parentObject->publicArea.nameAlg;
575
576             // Copy parent name
577             outCreation->creationData.parentName = parentObject->name;
578
579             // Copy parent qualified name
580             outCreation->creationData.parentQualifiedName = parentObject->qualifiedName;
581         }
582         // Copy outside information
583         outCreation->creationData.outsideInfo = *outsideData;
584
585         // Marshal creation data to canonical form
586         buffer = creationBuffer;
587         outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
588                                                        &buffer, NULL);
589         // Compute hash for creation field in public template
590         creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
591         CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
592         CryptHashEnd2B(&hashState, &creationDigest->b);
593
594         return;
595     }
```

7.6.3.7 GetSeedForKDF()

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

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

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

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

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

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

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

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

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.

```
752     static UINT16
753     MarshalSensitive(
754         OBJECT              *parent,            // IN: the object parent (optional)
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     #if ALG_RSA
770         // If the sensitive size is the special case for a prime in the type
771         if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) > 0)
772         {
773             UINT16              sizeSave = sensitive->sensitive.rsa.t.size;
774         //
775             // Turn off the flag that indicates that the sensitive->sensitive contains
776             // the CRT form of the exponent.
777             sensitive->sensitive.rsa.t.size &= ~(RSA_prime_flag);
778             // If the parent isn't fixedTPM, then truncate the sensitive data to be
779             // the size of the prime. Otherwise, leave it at the current size which
780             // is the full CRT size.
781             if(parent == NULL
782                || !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
783                                 TPMA_OBJECT, fixedTPM))
784                 sensitive->sensitive.rsa.t.size /= 5;
785             retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
786             // Restore the flag and the size.
787             sensitive->sensitive.rsa.t.size = sizeSave;
788         }
789         else
790     #endif
791         retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
792
793         // Marshal the size
794         retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
795
796         return retVal;
797     }
```

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.

```
798     void
799     SensitiveToPrivate(
800         TPMT_SENSITIVE  *sensitive,     // IN: sensitive structure
801         TPM2B_NAME      *name,          // IN: the name of the object
802         OBJECT          *parent,        // IN: The parent object
803         TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  This
804                                         //     parameter is used when parentHandle is
805                                         //     NULL, in which case the object is
806                                         //     temporary.
807         TPM2B_PRIVATE   *outPrivate     // OUT: output private structure
808         )
809     {
810         BYTE                *sensitiveData;     // pointer to the sensitive data
811         UINT16              dataSize;           // data blob size
812         TPMI_ALG_HASH       hashAlg;            // hash algorithm for integrity
813         UINT16              integritySize;
814         UINT16              ivSize;
815     //
816         pAssert(name != NULL && name->t.size != 0);
817
818         // Find the hash algorithm for integrity computation
819         if(parent == NULL)
```

```
820         {
821             // For Temporary Object, using self name algorithm
822             hashAlg = nameAlg;
823         }
824         else
825         {
826             // Otherwise, using parent's name algorithm
827             hashAlg = parent->publicArea.nameAlg;
828         }
829         // Starting of sensitive data without wrappers
830         sensitiveData = outPrivate->t.buffer;
831 
832         // Compute the integrity size
833         integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
834 
835         // Reserve space for integrity
836         sensitiveData += integritySize;
837 
838         // Get iv size
839         ivSize = GetIV2BSize(parent);
840 
841         // Reserve space for iv
842         sensitiveData += ivSize;
843 
844         // Marshal the sensitive area including authValue size adjustments.
845         dataSize = MarshalSensitive(parent, sensitiveData, sensitive, nameAlg);
846 
847         //Produce outer wrap, including encryption and HMAC
848         outPrivate->t.size = ProduceOuterWrap(parent, &name->b, hashAlg, NULL,
849                                               TRUE, dataSize, outPrivate->t.buffer);
850         return;
851     }
```

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

```
852     TPM_RC
853     PrivateToSensitive(
854         TPM2B           *inPrivate,     // IN: input private structure
855         TPM2B           *name,          // IN: the name of the object
856         OBJECT          *parent,        // IN: parent object
857         TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  It is
858                                         //     passed separately because we only pass
859                                         //     name, rather than the whole public area
860                                         //     of the object.  This parameter is used in
861                                         //     the following two cases: 1. primary
862                                         //     objects. 2. duplication blob with inner
```

```
863                                     //     wrap.  In other cases, this parameter
864                                     //     will be ignored
865      TPMT_SENSITIVE  *sensitive     // OUT: sensitive structure
866      )
867  {
868      TPM_RC          result;
869      BYTE            *buffer;
870      INT32           size;
871      BYTE            *sensitiveData; // pointer to the sensitive data
872      UINT16          dataSize;
873      UINT16          dataSizeInput;
874      TPMI_ALG_HASH   hashAlg;        // hash algorithm for integrity
875      UINT16          integritySize;
876      UINT16          ivSize;
877  //
878      // Make sure that name is provided
879      pAssert(name != NULL && name->size != 0);
880
881      // Find the hash algorithm for integrity computation
882      // For Temporary Object (parent == NULL) use self name algorithm;
883      // Otherwise, using parent's name algorithm
884      hashAlg = (parent == NULL) ? nameAlg : parent->publicArea.nameAlg;
885
886      // unwrap outer
887      result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
888                           inPrivate->size, inPrivate->buffer);
889      if(result != TPM_RC_SUCCESS)
890          return result;
891      // Compute the inner integrity size.
892      integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
893
894      // Get iv size
895      ivSize = GetIV2BSize(parent);
896
897      // The starting of sensitive data and data size without outer wrapper
898      sensitiveData = inPrivate->buffer + integritySize + ivSize;
899      dataSize = inPrivate->size - integritySize - ivSize;
900
901      // Unmarshal input data size
902      buffer = sensitiveData;
903      size = (INT32)dataSize;
904      result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
905      if(result == TPM_RC_SUCCESS)
906      {
907          if((dataSizeInput + sizeof(UINT16)) != dataSize)
908              result = TPM_RC_SENSITIVE;
909          else
910          {
911              // Unmarshal sensitive buffer to sensitive structure
912              result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
913              if(result != TPM_RC_SUCCESS || size != 0)
914              {
915                  result = TPM_RC_SENSITIVE;
916              }
917          }
918      }
919      return result;
920  }
```

7.6.3.13 SensitiveToDuplicate()

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

a) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply inner wrap to the sensitive area if required

c) apply outer wrap if required

```
void
SensitiveToDuplicate(
    TPMT_SENSITIVE      *sensitive,     // IN: sensitive structure
    TPM2B               *name,          // IN: the name of the object
    OBJECT              *parent,        // IN: The new parent object
    TPM_ALG_ID           nameAlg,       // IN: hash algorithm in public area. It
                                        //     is passed separately because we
                                        //     only pass name, rather than the
                                        //     whole public area of the object.
    TPM2B               *seed,          // IN: the external seed. If external
                                        //     seed is provided with size of 0,
                                        //     no outer wrap should be applied
                                        //     to duplication blob.
    TPMT_SYM_DEF_OBJECT *symDef,        // IN: Symmetric key definition. If the
                                        //     symmetric key algorithm is NULL,
                                        //     no inner wrap should be applied.
    TPM2B_DATA          *innerSymKey,   // IN/OUT: a symmetric key may be
                                        //     provided to encrypt the inner
                                        //     wrap of a duplication blob. May
                                        //     be generated here if needed.
    TPM2B_PRIVATE       *outPrivate     // OUT: output private structure
    )
{
    BYTE            *sensitiveData; // pointer to the sensitive data
    TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
    TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
    UINT16          dataSize;       // data blob size
    BOOL            doInnerWrap = FALSE;
    BOOL            doOuterWrap = FALSE;
//
    // Make sure that name is provided
    pAssert(name != NULL && name->size != 0);

    // Make sure symDef and innerSymKey are not NULL
    pAssert(symDef != NULL && innerSymKey != NULL);

    // Starting of sensitive data without wrappers
    sensitiveData = outPrivate->t.buffer;

    // Find out if inner wrap is required
    if(symDef->algorithm != TPM_ALG_NULL)
    {
        doInnerWrap = TRUE;

        // Use self nameAlg as inner hash algorithm
        innerHash = nameAlg;

        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
    }
    // Find out if outer wrap is required
    if(seed->size != 0)
    {
        doOuterWrap = TRUE;

        // Use parent nameAlg as outer hash algorithm
        outerHash = parent->publicArea.nameAlg;

        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
```

```
    }
    // Marshal sensitive area
    dataSize = MarshalSensitive(NULL, sensitiveData, sensitive, nameAlg);

    // Apply inner wrap for duplication blob.  It includes both integrity and
    // encryption
    if(doInnerWrap)
    {
        BYTE            *innerBuffer = NULL;
        BOOL            symKeyInput = TRUE;
        innerBuffer = outPrivate->t.buffer;
        // Skip outer integrity space
        if(doOuterWrap)
            innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
        dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
                                         innerBuffer);
        // Generate inner encryption key if needed
        if(innerSymKey->t.size == 0)
        {
            innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
            CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);

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

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

        // If the symmetric encryption key is imported, clear the buffer for
        // output
        if(symKeyInput)
            innerSymKey->t.size = 0;
    }
    // Apply outer wrap for duplication blob.  It includes both integrity and
    // encryption
    if(doOuterWrap)
    {
        dataSize = ProduceOuterWrap(parent, name, outerHash, seed, FALSE,
                                    dataSize, outPrivate->t.buffer);
    }
    // Data size for output
    outPrivate->t.size = dataSize;

    return;
}
```

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

```
1034  TPM_RC
1035  DuplicateToSensitive(
1036      TPM2B                *inPrivate,     // IN: input private structure
1037      TPM2B                *name,          // IN: the name of the object
1038      OBJECT               *parent,        // IN: the parent
1039      TPM_ALG_ID            nameAlg,       // IN: hash algorithm in public area.
1040      TPM2B                *seed,          // IN: an external seed may be provided.
1041                                           //     If external seed is provided with
1042                                           //     size of 0, no outer wrap is
1043                                           //     applied
1044      TPMT_SYM_DEF_OBJECT *symDef,         // IN: Symmetric key definition. If the
1045                                           //     symmetric key algorithm is NULL,
1046                                           //     no inner wrap is applied
1047      TPM2B                *innerSymKey,   // IN: a symmetric key may be provided
1048                                           //     to decrypt the inner wrap of a
1049                                           //     duplication blob.
1050      TPMT_SENSITIVE       *sensitive      // OUT: sensitive structure
1051      )
1052  {
1053      TPM_RC                result;
1054      BYTE                 *buffer;
1055      INT32                 size;
1056      BYTE                 *sensitiveData; // pointer to the sensitive data
1057      UINT16                dataSize;
1058      UINT16                dataSizeInput;
1059  //
1060      // Make sure that name is provided
1061      pAssert(name != NULL && name->size != 0);
1062
1063      // Make sure symDef and innerSymKey are not NULL
1064      pAssert(symDef != NULL && innerSymKey != NULL);
1065
1066      // Starting of sensitive data
1067      sensitiveData = inPrivate->buffer;
1068      dataSize = inPrivate->size;
1069
1070      // Find out if outer wrap is applied
1071      if(seed->size != 0)
1072      {
1073          // Use parent nameAlg as outer hash algorithm
1074          TPMI_ALG_HASH   outerHash = parent->publicArea.nameAlg;
1075
1076          result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
1077                               dataSize, sensitiveData);
1078          if(result != TPM_RC_SUCCESS)
1079              return result;
1080          // Adjust sensitive data pointer and size
1081          sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1082          dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1083      }
1084      // Find out if inner wrap is applied
1085      if(symDef->algorithm != TPM_ALG_NULL)
```

```
1086        {
1087            // assume the input key size matches the symmetric definition
1088            pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
1089
1090            // Decrypt inner buffer in place
1091            CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1092                                  symDef->keyBits.sym, innerSymKey->buffer, NULL,
1093                                  TPM_ALG_CFB, dataSize, sensitiveData);
1094            // Check inner integrity
1095            result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
1096            if(result != TPM_RC_SUCCESS)
1097                return result;
1098            // Adjust sensitive data pointer and size
1099            sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1100            dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1101        }
1102        // Unmarshal input data size
1103        buffer = sensitiveData;
1104        size = (INT32)dataSize;
1105        result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1106        if(result == TPM_RC_SUCCESS)
1107        {
1108            if((dataSizeInput + sizeof(UINT16)) != dataSize)
1109                result = TPM_RC_SIZE;
1110            else
1111            {
1112                // Unmarshal sensitive buffer to sensitive structure
1113                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1114
1115                // if the results is OK make sure that all the data was unmarshaled
1116                if(result == TPM_RC_SUCCESS && size != 0)
1117                    result = TPM_RC_SIZE;
1118            }
1119        }
1120        return result;
1121    }
```

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

```
1122    void
1123    SecretToCredential(
1124        TPM2B_DIGEST        *secret,        // IN: secret information
1125        TPM2B               *name,          // IN: the name of the object
1126        TPM2B               *seed,          // IN: an external seed.
1127        OBJECT              *protector,     // IN: the protector
1128        TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
1129        )
1130    {
1131        BYTE                *buffer;        // Auxiliary buffer pointer
1132        BYTE                *sensitiveData; // pointer to the sensitive data
1133        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1134        UINT16               dataSize;      // data blob size
1135    //
1136        pAssert(secret != NULL && outIDObject != NULL);
```

```
1137
1138        // use protector's name algorithm as outer hash ????
1139        outerHash = protector->publicArea.nameAlg;
1140
1141        // Marshal secret area to credential buffer, leave space for integrity
1142        sensitiveData = outIDObject->t.credential
1143            + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1144    // Marshal secret area
1145        buffer = sensitiveData;
1146        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1147
1148        // Apply outer wrap
1149        outIDObject->t.size = ProduceOuterWrap(protector, name, outerHash, seed, FALSE,
1150                                               dataSize, outIDObject->t.credential);
1151        return;
1152    }
```

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

```
1153    TPM_RC
1154    CredentialToSecret(
1155        TPM2B               *inIDObject,    // IN: input credential blob
1156        TPM2B               *name,          // IN: the name of the object
1157        TPM2B               *seed,          // IN: an external seed.
1158        OBJECT              *protector,     // IN: the protector
1159        TPM2B_DIGEST        *secret         // OUT: secret information
1160        )
1161    {
1162        TPM_RC                   result;
1163        BYTE                    *buffer;
1164        INT32                    size;
1165        TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
1166        BYTE                    *sensitiveData; // pointer to the sensitive data
1167        UINT16                   dataSize;
1168    //
1169        // use protector's name algorithm as outer hash
1170        outerHash = protector->publicArea.nameAlg;
1171
1172        // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1173        result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1174                             inIDObject->size, inIDObject->buffer);
1175        if(result == TPM_RC_SUCCESS)
1176        {
1177            // Compute the beginning of sensitive data
1178            sensitiveData = inIDObject->buffer
1179                + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1180            dataSize = inIDObject->size
```

```
1181                - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
1182            // Unmarshal secret buffer to TPM2B_DIGEST structure
1183            buffer = sensitiveData;
1184            size = (INT32)dataSize;
1185            result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1186
1187            // If there were no other unmarshaling errors, make sure that the
1188            // expected amount of data was recovered
1189            if(result == TPM_RC_SUCCESS && size != 0)
1190                return TPM_RC_SIZE;
1191        }
1192        return result;
1193    }
```

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.

```
1194    UINT16
1195    MemoryRemoveTrailingZeros(
1196        TPM2B_AUTH      *auth           // IN/OUT: value to adjust
1197        )
1198    {
1199        while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1200            auth->t.size--;
1201        return auth->t.size;
1202    }
```

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.

```
1203    TPM_RC
1204    SetLabelAndContext(
1205        TPMS_DERIVE             *labelContext,  // IN/OUT: the recovered label and
1206                                                //      context
1207        TPM2B_SENSITIVE_DATA    *sensitive      // IN: the sensitive data
1208        )
1209    {
1210        TPMS_DERIVE              sensitiveValue;
1211        TPM_RC                   result;
1212        INT32                    size;
1213        BYTE                    *buff;
1214    //
1215        // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1216        // If there is something to unmarshal...
1217        if(sensitive->t.size != 0)
1218        {
1219            size = sensitive->t.size;
1220            buff = sensitive->t.buffer;
1221            result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
1222            if(result != TPM_RC_SUCCESS)
1223                return result;
1224            // If there was a label in the public area leave it there, otherwise, copy
1225            // the new value
1226            if(labelContext->label.t.size == 0)
1227                MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
1228                             sizeof(labelContext->label.t.buffer));
1229            // if there was a context string in publicArea, it overrides
```

```
1230            if(labelContext->context.t.size == 0)
1231                MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
1232                             sizeof(labelContext->label.t.buffer));
1233        }
1234        return TPM_RC_SUCCESS;
1235    }
```

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*.

```
1236    TPM_RC
1237    UnmarshalToPublic(
1238        TPMT_PUBLIC         *tOut,       // OUT: output
1239        TPM2B_TEMPLATE      *tIn,        // IN:
1240        BOOL                 derivation, // IN: indicates if this is for a derivation
1241        TPMS_DERIVE         *labelContext// OUT: label and context if derivation
1242        )
1243    {
1244        BYTE                *buffer = tIn->t.buffer;
1245        INT32                size = tIn->t.size;
1246        TPM_RC               result;
1247    //
1248        // make sure that tOut is zeroed so that there are no remnants from previous
1249        // uses
1250        MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1251        // Unmarshal the components of the TPMT_PUBLIC up to the unique field
1252        result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
1253        if(result != TPM_RC_SUCCESS)
1254            return result;
1255        result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
1256        if(result != TPM_RC_SUCCESS)
1257            return result;
1258        result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
1259        if(result != TPM_RC_SUCCESS)
1260            return result;
1261        result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
1262        if(result != TPM_RC_SUCCESS)
1263            return result;
1264        result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size,
1265                                             tOut->type);
1266        if(result != TPM_RC_SUCCESS)
1267            return result;
1268        // Now unmarshal a TPMS_DERIVE if this is for derivation
1269        if(derivation)
1270            result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
1271        else
1272            // otherwise, unmarshal a TPMU_PUBLIC_ID
1273            result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size,
1274                                              tOut->type);
1275        // Make sure the template was used up
1276        if((result == TPM_RC_SUCCESS) && (size != 0))
1277            result = TPM_RC_SIZE;
1278        return result;
1279    }
```

7.6.3.20 ObjectSetExternal()

Set the external attributes for an object.

```
1280    void
1281    ObjectSetExternal(
1282        OBJECT      *object
1283        )
1284    {
1285        object->attributes.external = SET;
1286    }
```

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
```

7.8 ACT Support (ACT_spt.c)

7.8.1 Introduction

This code implements the ACT update code. It does not use a mutex. This code uses a platform service (_plat__ACT_UpdateCounter()) that returns *false* if the update is not accepted. If this occurs, then TPM_RC_RETRY should be sent to the caller so that they can retry the operation later. The implementation of this is platform dependent but the reference uses a simple flag to indicate that an update is pending and the only process that can clear that flag is the process that does the actual update.

7.8.2 Includes

```
#include "Tpm.h"
#include "ACT_spt_fp.h"
#include "Platform_fp.h"
```

7.8.3 Functions

7.8.3.1 _ActResume()

This function does the resume processing for an ACT. It updates the saved count and turns signaling back on if necessary.

```
static void
_ActResume(
    UINT32              act,            //IN: the act number
    ACT_STATE          *actData         //IN: pointer to the saved ACT data
)
{
    // If the act was non-zero, then restore the counter value.
    if(actData->remaining > 0)
        _plat__ACT_UpdateCounter(act, actData->remaining);
    // if the counter was zero and the ACT signaling, enable the signaling.
    else if(go.signaledACT & (1 << act))
        _plat__ACT_SetSignaled(act, TRUE);
}
```

7.8.3.2 ActStartup()

This function is called by TPM2_Startup() to initialize the ACT counter values.

```
BOOL
ActStartup(
    STARTUP_TYPE        type
)
{
    // Reset all the ACT hardware
    _plat__ACT_Initialize();

    // For TPM_RESET or TPM_RESTART, the ACTs will all be disabled and the output
    // de-asserted.
    if(type != SU_RESUME)
    {
        go.signaledACT = 0;
#define CLEAR_ACT_POLICY(N)                                                    \
        go.ACT_##N.hashAlg = TPM_ALG_NULL;                                     \
        go.ACT_##N.authPolicy.b.size = 0;

```

```
34          FOR_EACH_ACT(CLEAR_ACT_POLICY)
35
36      }
37      else
38      {
39          // Resume each of the implemented ACT
40  #define RESUME_ACT(N)   _ActResume(0x##N, &go.ACT_##N);
41
42          FOR_EACH_ACT(RESUME_ACT)
43      }
44      s_ActUpdated = 0;
45      _plat__ACT_EnableTicks(TRUE);
46      return TRUE;
47  }
```

7.8.3.3 _ActSaveState()

Get the counter state and the signaled state for an ACT. If the ACT has not been updated since the last time it was saved, then divide the count by 2.

```
48  static void
49  _ActSaveState(
50      UINT32              act,
51      P_ACT_STATE         actData
52  )
53  {
54      actData->remaining = _plat__ACT_GetRemaining(act);
55      // If the ACT hasn't been updated since the last startup, then it should be
56      // be halved.
57      if((s_ActUpdated & (1 << act)) == 0)
58      {
59          // Don't halve if the count is set to max or if halving would make it zero
60          if((actData->remaining != UINT32_MAX) && (actData->remaining > 1))
61              actData->remaining /= 2;
62      }
63      if(_plat__ACT_GetSignaled(act))
64          go.signaledACT |= (1 << act);
65  }
```

7.8.3.4 ActGetSignaled()

This function returns the state of the signaled flag associated with an ACT.

```
66  BOOL
67  ActGetSignaled(
68      TPM_RH              actHandle
69  )
70  {
71      UINT32              act = actHandle - TPM_RH_ACT_0;
72  //
73      return _plat__ACT_GetSignaled(act);
74  }
```

7.8.3.5 ActShutdown()

This function saves the current state of the counters

```
75  BOOL
76  ActShutdown(
77      TPM_SU              state       //IN: the type of the shutdown.
78  )
```

```
{
    // if this is not shutdown state, then the only type of startup is TPM_RESTART
    // so the timer values will be cleared. If this is shutdown state, get the current
    // countdown and signaled values. Plus, if the counter has not been updated
    // since the last restart, divide the time by 2 so that there is no attack on the
    // countdown by saving the countdown state early and then not using the TPM.
    if(state == TPM_SU_STATE)
    {
        // This will be populated as each of the ACT is queried
        go.signaledACT = 0;
        // Get the current count and the signaled state
#define SAVE_ACT_STATE(N) _ActSaveState(0x##N, &go.ACT_##N);

        FOR_EACH_ACT(SAVE_ACT_STATE);
    }
    return TRUE;
}
```

7.8.3.6 ActIsImplemented()

This function determines if an ACT is implemented in both the TPM and the platform code.

```
BOOL
ActIsImplemented(
    UINT32          act
)
{
#define CASE_ACT_
    // This switch accounts for the TPM implemente values.
    switch(act)
    {
        FOR_EACH_ACT(CASE_ACT_NUMBER)
            // This ensures that the platorm implementes the values implemented by
            // the TPM
            return _plat__ACT_GetImplemented(act);
        default:
            break;
    }
    return FALSE;
}
```

7.8.3.7 ActCounterUpdate()

This function updates the ACT counter. If the counter already has a pending update, it returns TPM_RC_RETRY so that the update can be tried again later.

```
TPM_RC
ActCounterUpdate(
    TPM_RH          handle,         //IN: the handle of the act
    UINT32          newValue        //IN: the value to set in the ACT
)
{
    UINT32          act;
    TPM_RC          result;
//
    act = handle - TPM_RH_ACT_0;
    // This should never fail, but...
    if(!_plat__ACT_GetImplemented(act))
        result = TPM_RC_VALUE;
    else
    {
    // Will need to clear orderly so fail if we are orderly and NV is not available
        if(NV_IS_ORDERLY)
```

```
131             RETURN_IF_NV_IS_NOT_AVAILABLE;
132         // if the attempt to update the counter fails, it means that there is an
133         // update pending so wait until it has occurred and then do an update.
134         if(!_plat__ACT_UpdateCounter(act, newValue))
135             result = TPM_RC_RETRY;
136         else
137         {
138             // Indicate that the ACT has been updated since last TPM2_Startup().
139             s_ActUpdated |= (UINT16)(1 << act);
140 
141             // Need to clear the orderly flag
142             g_clearOrderly = TRUE;
143 
144             result = TPM_RC_SUCCESS;
145         }
146     }
147     return result;
148 }
```

7.8.3.8 ActGetCapabilityData()

This function returns the list of ACT data

Return Value	Meaning
YES	if more ACT data is available
NO	if no more ACT data to

```
149 TPMI_YES_NO
150 ActGetCapabilityData(
151     TPM_HANDLE       actHandle,     // IN: the handle for the starting ACT
152     UINT32           maxCount,      // IN: maximum allowed return values
153     TPML_ACT_DATA   *actList        // OUT: ACT data list
154 )
155 {
156     // Initialize output property list
157     actList->count = 0;
158 
159     // Make sure that the starting handle value is in range (again)
160     if((actHandle < TPM_RH_ACT_0) || (actHandle > TPM_RH_ACT_F))
161         return FALSE;
162     // The maximum count of curves we may return is MAX_ECC_CURVES
163     if(maxCount > MAX_ACT_DATA)
164         maxCount = MAX_ACT_DATA;
165     // Scan the ACT data from the starting ACT
166     for(; actHandle <= TPM_RH_ACT_F; actHandle++)
167     {
168         UINT32          act = actHandle - TPM_RH_ACT_0;
169         if(actList->count < maxCount)
170         {
171             if(ActIsImplemented(act))
172             {
173                 TPMS_ACT_DATA    *actData = &actList->actData[actList->count];
174                 //
175                 memset(&actData->attributes, 0, sizeof(actData->attributes));
176                 actData->handle = actHandle;
177                 actData->timeout = _plat__ACT_GetRemaining(act);
178                 actData->attributes.signaled = _plat__ACT_GetSignaled(act);
179                 actList->count++;
180             }
181         }
182         else
183         {
```

```
184                 if(_plat__ACT_GetImplemented(act))
185                     return YES;
186             }
187         }
188         // If we get here, either all of the ACT values were put in the list, or the list
189         // was filled and there are no more ACT values to return
190         return NO;
191     }
```

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 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.

```
BOOL
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      return TRUE;
38  }
```

8.1.3.3 CommandAuditSet()

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

This function is only used by TPM2_SetCommandCodeAuditStatus().

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

Return Value	Meaning
TRUE(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```
39  BOOL
40  CommandAuditSet(
41      TPM_CC            commandCode     // IN: command code
42      )
43  {
44      COMMAND_INDEX         commandIndex = CommandCodeToCommandIndex(commandCode);
45
46      // Only SET a bit if the corresponding command is implemented
47      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
48      {
49          // Can't audit shutdown
50          if(commandCode != TPM_CC_Shutdown)
51          {
52              if(!TEST_BIT(commandIndex, gp.auditCommands))
53              {
54                  // Set bit
55                  SET_BIT(commandIndex, gp.auditCommands);
56                  return TRUE;
57              }
58          }
59      }
60      // No change
61      return FALSE;
62  }
```

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

```
63  BOOL
64  CommandAuditClear(
65      TPM_CC            commandCode    // IN: command code
66      )
67  {
68      COMMAND_INDEX       commandIndex = CommandCodeToCommandIndex(commandCode);
69
70      // Do nothing if the command is not implemented
71      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
72      {
73          // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
74          // cleared
75          if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
76          {
77              if(TEST_BIT(commandIndex, gp.auditCommands))
78              {
79                  // Clear bit
80                  CLEAR_BIT(commandIndex, gp.auditCommands);
81                  return TRUE;
82              }
83          }
84      }
85      // No change
86      return FALSE;
87  }
```

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

```
88  BOOL
89  CommandAuditIsRequired(
90      COMMAND_INDEX     commandIndex   // IN: command index
91      )
92  {
93      // Check the bit map.  If the bit is SET, command audit is required
94      return(TEST_BIT(commandIndex, gp.auditCommands));
95  }
```

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(
```

```
146         TPM2B_DIGEST    *digest         // OUT: command digest
147         )
148     {
149         TPM_CC                      commandCode;
150         COMMAND_INDEX               commandIndex;
151         HASH_STATE                  hashState;
152 
153         // Start hash
154         digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
155 
156         // Add command code
157         for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
158         {
159             if(CommandAuditIsRequired(commandIndex))
160             {
161                 commandCode = GetCommandCode(commandIndex);
162                 CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
163             }
164         }
165 
166         // Complete hash
167         CryptHashEnd2B(&hashState, &digest->b);
168 
169         return;
170     }
```

8.2 DA.c

8.2.1 Introduction

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

8.2.2 Includes and Data Definitions

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

8.2.3 Functions

8.2.3.1 DAPreInstall_Init()

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

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

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

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

    // Record persistent DA parameter changes to NV
    NV_SYNC_PERSISTENT(failedTries);
    NV_SYNC_PERSISTENT(maxTries);
    NV_SYNC_PERSISTENT(recoveryTime);
    NV_SYNC_PERSISTENT(lockoutRecovery);
    NV_SYNC_PERSISTENT(lockOutAuthEnabled);

    return;
}
```

8.2.3.2 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.

```
BOOL
DAStartup(
    STARTUP_TYPE     type            // IN: startup type
    )
{
    NOT_REFERENCED(type);
#if !ACCUMULATE_SELF_HEAL_TIMER
    _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 TRUE;
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          )
```

```
{
    // Reset the timer associated with lockout if the handle is the lockoutAuth.
    if(handle == TPM_RH_LOCKOUT)
        s_lockoutTimer = g_time;
    else
        s_selfHealTimer = g_time;
    return;
}
```

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.

```
void
DASelfHeal(
    void
    )
{
    // Regular authorization self healing logic
    // If no failed authorization tries, do nothing.  Otherwise, try to
    // decrease failedTries
    if(gp.failedTries != 0)
    {
        // if recovery time is 0, DA logic has been disabled.  Clear failed tries
        // immediately
        if(gp.recoveryTime == 0)
        {
            gp.failedTries = 0;
            // Update NV record
            NV_SYNC_PERSISTENT(failedTries);
        }
        else
        {
            UINT64          decreaseCount;
#if 0 // Errata eliminates this code
            // In the unlikely event that failedTries should become larger than
            // maxTries
            if(gp.failedTries > gp.maxTries)
                gp.failedTries = gp.maxTries;
#endif
            // How much can failedTries be decreased

            // Cast s_selfHealTimer to an int in case it became negative at
            // startup
            decreaseCount = ((g_time - (INT64)s_selfHealTimer) / 1000)
                / gp.recoveryTime;

            if(gp.failedTries <= (UINT32)decreaseCount)
                // should not set failedTries below zero
                gp.failedTries = 0;
            else
                gp.failedTries -= (UINT32)decreaseCount;

            // the cast prevents overflow of the product
            s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
            if(decreaseCount != 0)
                // If there was a change to the failedTries, record the changes
                // to NV
                NV_SYNC_PERSISTENT(failedTries);
        }
    }
```

```
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  BOOL
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          gc.platformAlg = TPM_ALG_NULL;
78
79          // enable the storage and endorsement hierarchies and the platformNV
80          gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
81      }
82
83      // nullProof and nullSeed are updated at every TPM_RESET
84      if((type != SU_RESTART) && (type != SU_RESUME))
85      {
86          gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
87          CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
88          gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
89          CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
90      }
91
92      return TRUE;
93  }
```

8.3.3.3 HierarchyGetProof()

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

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

```
    )
{
    TPM2B_PROOF         *proof = NULL;

    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
            // phProof for TPM_RH_PLATFORM
            proof = &gp.phProof;
            break;
        case TPM_RH_ENDORSEMENT:
            // ehProof for TPM_RH_ENDORSEMENT
            proof = &gp.ehProof;
            break;
        case TPM_RH_OWNER:
            // shProof for TPM_RH_OWNER
            proof = &gp.shProof;
            break;
        default:
            // nullProof for TPM_RH_NULL or anything else
            proof = &gr.nullProof;
            break;
    }
    return proof;
}
```

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

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

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

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

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

```
1   #define NV_C
2   #include "Tpm.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.

```
3   static NV_REF
4   NvNext(
5       NV_REF          *iter,          // IN/OUT: the list iterator
6       TPM_HANDLE      *handle         // OUT: the handle of the next item.
7       )
8   {
9       NV_REF                currentAddr;
10      NV_ENTRY_HEADER       header;
11  //
12      // If iterator is at the beginning of list
```

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

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

```
35  static NV_REF
36  NvNextByType(
37      TPM_HANDLE      *handle,        // OUT: the handle of the found type
38      NV_REF          *iter,          // IN: the iterator
39      TPM_HT           type           // IN: the handle type to look for
40      )
41  {
42      NV_REF           addr;
43      TPM_HANDLE       nvHandle;
44  //
45      while((addr = NvNext(iter, &nvHandle)) != 0)
46      {
47          // addr: the address of the location containing the handle of the value
48          // iter: the next location.
49          if(HandleGetType(nvHandle) == type)
50              break;
51      }
52      if(handle != NULL)
53          *handle = nvHandle;
54      return addr;
55  }
```

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

```
static BOOL
NvTestSpace(
    UINT32           size,          // IN: size of the entity to be added
    BOOL             isIndex,       // IN: TRUE if the entity is an index
    BOOL             isCounter      // IN: TRUE if the index is a counter
    )
{
    UINT32      remainBytes = NvGetFreeBytes();
    UINT32      reserved = sizeof(UINT32)       // size of the forward pointer
        + sizeof(NV_LIST_TERMINATOR);
//
    // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
#if NV_MEMORY_SIZE < 1024
#error "NV_MEMORY_SIZE probably isn't large enough"
#endif

    // For NV Index, need to make sure that we do not allocate an Index if this
    // would mean that the TPM cannot allocate the minimum number of evict
    // objects.
    if(isIndex)
    {
        // Get the number of persistent objects allocated
        UINT32      persistentNum = NvCapGetPersistentNumber();

        // If we have not allocated the requisite number of evict objects, then we
        // need to reserve space for them.
        // NOTE: some of this is not written as simply as it might seem because
        // the values are all unsigned and subtracting needs to be done carefully
        // so that an underflow doesn't cause problems.
        if(persistentNum < MIN_EVICT_OBJECTS)
            reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
    }
    // If this is not an index or is not a counter, reserve space for the
    // required number of counter indexes
    if(!isIndex || !isCounter)
    {
        // Get the number of counters
        UINT32      counterNum = NvCapGetCounterNumber();

        // If the required number of counters have not been allocated, reserved
        // space for the extra needed counters
        if(counterNum < MIN_COUNTER_INDICES)
            reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
    }
    // Check that the requested allocation will fit after making sure that there
    // will be no chance of overflow
    return ((reserved < remainBytes)
            && (size <= remainBytes)
            && (size + reserved <= remainBytes));
}
```

8.4.3.8 NvWriteNvListEnd()

Function to write the list terminator.

```
NV_REF
NvWriteNvListEnd(
    NV_REF           end
```

```
136         )
137     {
138         // Marker is initialized with zeros
139         BYTE        listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
140         UINT64      maxCount = NvReadMaxCount();
141     //
142         // This is a constant check that can be resolved at compile time.
143         cAssert(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
144
145         // Copy the maxCount value to the marker buffer
146         MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
147         pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
148
149         // Write it to memory
150         NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
151         return end + sizeof(NV_LIST_TERMINATOR);
152     }
```

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.

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

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

8.4.3.10 NvDelete()

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

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

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

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

8.4.4.3 NvRamGetEnd()

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

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

```
{
    NV_RAM_REF           iter = NV_RAM_REF_INIT;
    NV_RAM_REF           currentAddr;
//
    // Scan until the next address is 0
    while((currentAddr = NvRamNext(&iter, NULL)) != 0);
    return iter;
}
```

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

```
static BOOL
NvRamTestSpaceIndex(
    UINT32            size           // IN: size of the data to be added to RAM
    )
{
    UINT32          remaining = (UINT32)(RAM_ORDERLY_END - NvRamGetEnd());
    UINT32          needed = sizeof(NV_RAM_HEADER) + size;
//
    // NvRamGetEnd points to the next available byte.
    return remaining >= needed;
}
```

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.

```
static NV_RAM_REF
NvRamGetIndex(
    TPMI_RH_NV_INDEX     handle          // IN: NV handle
    )
{
    NV_RAM_REF          iter = NV_RAM_REF_INIT;
    NV_RAM_REF          currentAddr;
    TPM_HANDLE          foundHandle;
//
    while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
    {
        if(handle == foundHandle)
            break;
    }
    return currentAddr;
}
```

8.4.4.6 NvUpdateIndexOrderlyData()

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

```
void
NvUpdateIndexOrderlyData(
    void
    )
```

```
326 {
327     // Write reserved RAM space to NV
328     NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
329 }
```

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.

```
330 static void
331 NvAddRAM(
332     TPMS_NV_PUBLIC  *index          // IN: the index descriptor
333     )
334 {
335     NV_RAM_HEADER       header;
336     NV_RAM_REF          end = NvRamGetEnd();
337 //
338     header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
339     header.handle = index->nvIndex;
340     MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
341
342     pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
343
344     // Copy the header to the memory
345     MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
346
347     // Clear the data area (just in case)
348     MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
349
350     // Step over this new entry
351     end += header.size;
352
353     // If the end marker will fit, add it
354     if(end + sizeof(UINT32) < RAM_ORDERLY_END)
355         MemorySet(end, 0, sizeof(UINT32));
356     // Write reserved RAM space to NV to reflect the newly added NV Index
357     SET_NV_UPDATE(UT_ORDERLY);
358
359     return;
360 }
```

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.

```
361 static void
362 NvDeleteRAM(
363     TPMI_RH_NV_INDEX     handle         // IN: NV handle
364     )
365 {
366     NV_RAM_REF           nodeAddress;
367     NV_RAM_REF           nextNode;
```

```
368         UINT32                size;
369         NV_RAM_REF            lastUsed = NvRamGetEnd();
370     //
371         nodeAddress = NvRamGetIndex(handle);
372
373         pAssert(nodeAddress != 0);
374
375         // Get node size
376         MemoryCopy(&size, nodeAddress, sizeof(size));
377
378         // Get the offset of next node
379         nextNode = nodeAddress + size;
380
381         // Copy the data
382         MemoryCopy(nodeAddress, nextNode, (int)(lastUsed - nextNode));
383
384         // Clear out the reclaimed space
385         MemorySet(lastUsed - size, 0, size);
386
387         // Write reserved RAM space to NV to reflect the newly delete NV Index
388         SET_NV_UPDATE(UT_ORDERLY);
389
390         return;
391     }
```

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.

```
392     void
393     NvReadNvIndexInfo(
394         NV_REF           ref,           // IN: points to NV where index is located
395         NV_INDEX        *nvIndex        // OUT: place to receive index data
396         )
397     {
398         pAssert(nvIndex != NULL);
399         NvRead(nvIndex, ref, sizeof(NV_INDEX));
400         return;
401     }
```

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.

```
402     void
403     NvReadObject(
404         NV_REF           ref,           // IN: points to NV where index is located
405         OBJECT          *object         // OUT: place to receive the object data
406         )
407     {
408         NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
409         return;
410     }
```

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

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

8.4.4.12 NvIndexIsDefined()

See if an index is already defined

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

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

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

8.4.4.14 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

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

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.

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

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

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

8.4.4.17 NvWriteRamIndexAttributes()

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

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

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

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

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

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

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

```
508  TPM_RC
509  NvIndexIsAccessible(
510      TPMI_RH_NV_INDEX     handle         // IN: handle
511      )
512  {
513      NV_INDEX            *nvIndex = NvGetIndexInfo(handle, NULL);
514  //
515      if(nvIndex == NULL)
516      // If index is not found, return TPM_RC_HANDLE
517          return TPM_RC_HANDLE;
518      if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
519      {
520          // if shEnable is CLEAR, an ownerCreate NV Index should not be
521          // indicated as present
522          if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
523          {
524              if(gc.shEnable == FALSE)
525                  return TPM_RC_HANDLE;
526          }
527          // if phEnableNV is CLEAR, a platform created Index should not
528          // be visible
529          else if(gc.phEnableNV == FALSE)
530              return TPM_RC_HANDLE;
531      }
532  #if 0 // Writelock test for debug
533      // If the Index is write locked and this is an NV Write operation...
534      if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
535         &&  IsWriteOperation(commandIndex))
536      {
537          // then return a locked indication unless the command is TPM2_NV_WriteLock
538          if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
539              return TPM_RC_NV_LOCKED;
540          return TPM_RC_SUCCESS;
541      }
542  #endif
543  #if 0   // Readlock Test for debug
544      // If the Index is read locked and this is an NV Read operation...
545      if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
546         && IsReadOperation(commandIndex))
547      {
548          // then return a locked indication unless the command is TPM2_NV_ReadLock
549          if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
550              return TPM_RC_NV_LOCKED;
551      }
552  #endif
553      // NV Index is accessible
554      return TPM_RC_SUCCESS;
555  }
```

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

```
556  TPM_RC
557  NvGetEvictObject(
558      TPM_HANDLE       handle,         // IN: handle
559      OBJECT          *object          // OUT: object data
560      )
561  {
562      NV_REF           entityAddr;          // offset points to the entity
563  //
564      // Find the address of evict object and copy to object
565      entityAddr = NvFindEvict(handle, object);
566
567      // whether there is an error or not, make sure that the evict
568      // status of the object is set so that the slot will get freed on exit
569      // Must do this after NvFindEvict loads the object
570      object->attributes.evict = SET;
571
572      // If handle is not found, return an error
573      if(entityAddr == 0)
574          return TPM_RC_HANDLE;
575      return TPM_RC_SUCCESS;
576  }
```

8.4.5.5 NvIndexCacheInit()

Function to initialize the Index cache

```
577  void
578  NvIndexCacheInit(
579      void
580      )
581  {
582      s_cachedNvRef = NV_REF_INIT;
583      s_cachedNvRamRef = NV_RAM_REF_INIT;
584      s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
585      return;
586  }
```

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.

```
587  void
588  NvGetIndexData(
589      NV_INDEX            *nvIndex,       // IN: the in RAM index descriptor
590      NV_REF               locator,       // IN: where the data is located
591      UINT32               offset,        // IN: offset of NV data
592      UINT16               size,          // IN: number of octets of NV data to read
593      void                *data           // OUT: data buffer
594      )
595  {
596      TPMA_NV             nvAttributes;
597  //
598      pAssert(nvIndex != NULL);
599
600      nvAttributes = nvIndex->publicArea.attributes;
```

```
601
602         pAssert(IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN));
603
604         if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, ORDERLY))
605         {
606             // Get data from RAM buffer
607             NV_RAM_REF           ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
608             pAssert(ramAddr != 0 && (size <=
609                     ((NV_RAM_HEADER *)ramAddr)->size - sizeof(NV_RAM_HEADER) - offset));
610             MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
611         }
612         else
613         {
614             // Validate that read falls within range of the index
615             pAssert(offset <= nvIndex->publicArea.dataSize
616                     &&  size <= (nvIndex->publicArea.dataSize - offset));
617             NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
618         }
619         return;
620     }
```

8.4.5.7 NvHashIndexData()

This function adds Index data to a hash. It does this in parts to avoid large stack buffers.

```
621     void
622     NvHashIndexData(
623         HASH_STATE          *hashState,     // IN: Initialized hash state
624         NV_INDEX            *nvIndex,       // IN: Index
625         NV_REF               locator,       // IN: where the data is located
626         UINT32               offset,        // IN: starting offset
627         UINT16               size           // IN: amount to hash
628     )
629     {
630     #define BUFFER_SIZE     64
631         BYTE                 buffer[BUFFER_SIZE];
632         if (offset > nvIndex->publicArea.dataSize)
633             return;
634         // Make sure that we don't try to read off the end.
635         if ((offset + size) > nvIndex->publicArea.dataSize)
636             size = nvIndex->publicArea.dataSize - (UINT16)offset;
637     #if BUFFER_SIZE >= MAX_NV_INDEX_SIZE
638         NvGetIndexData(nvIndex, locator, offset, size, buffer);
639         CryptDigestUpdate(hashState, size, buffer);
640     #else
641         {
642             INT16               i;
643             UINT16              readSize;
644             //
645             for (i = size; i > 0; offset += readSize, i -= readSize)
646             {
647                 readSize = (i < BUFFER_SIZE) ? i : BUFFER_SIZE;
648                 NvGetIndexData(nvIndex, locator, offset, readSize, buffer);
649                 CryptDigestUpdate(hashState, readSize, buffer);
650             }
651         }
652     #endif // BUFFER_SIZE >= MAX_NV_INDEX_SIZE
653     #undef  BUFFER_SIZE
654     }
```

8.4.5.8 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.

```
655 UINT64
656 NvGetUINT64Data(
657     NV_INDEX            *nvIndex,       // IN: the in RAM index descriptor
658     NV_REF               locator        // IN: where index exists in NV
659     )
660 {
661     UINT64                intVal;
662 //
663     // Read the value and convert it to internal format
664     NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
665     return BYTE_ARRAY_TO_UINT64(((BYTE *)&intVal));
666 }
```

8.4.5.9 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

```
667 TPM_RC
668 NvWriteIndexAttributes(
669     TPM_HANDLE        handle,
670     NV_REF            locator,       // IN: location of the index
671     TPMA_NV           attributes     // IN: attributes to write
672     )
673 {
674     TPM_RC              result;
675 //
676     if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
677     {
678         NV_RAM_REF      ram = NvRamGetIndex(handle);
679         NvWriteRamIndexAttributes(ram, attributes);
680         result = TPM_RC_SUCCESS;
681     }
682     else
683     {
684         result = NvWriteNvIndexAttributes(locator, attributes);
685     }
686     return result;
687 }
```

8.4.5.10 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

```
688 TPM_RC
689 NvWriteIndexAuth(
690     NV_REF            locator,       // IN: location of the index
691     TPM2B_AUTH       *authValue      // IN: the authValue to write
692     )
```

```
693 {
694     TPM_RC              result;
695 //
696     // If the locator is pointing to the cached index value...
697     if(locator == s_cachedNvRef)
698     {
699         // copy the authValue to the cached index so it will be there if we
700         // look for it. This is a safety thing.
701         MemoryCopy2B(&s_cachedNvIndex.authValue.b, &authValue->b,
702                      sizeof(s_cachedNvIndex.authValue.t.buffer));
703     }
704     result = NvConditionallyWrite(
705         locator + offsetof(NV_INDEX, authValue),
706         sizeof(UINT16) + authValue->t.size,
707         authValue);
708     return result;
709 }
```

8.4.5.11 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

```
710 NV_INDEX *
711 NvGetIndexInfo(
712     TPM_HANDLE       nvHandle,      // IN: the index handle
713     NV_REF          *locator        // OUT: location of the index
714     )
715 {
716     if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
717     {
718         s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
719         s_cachedNvRamRef = 0;
720         s_cachedNvRef = NvFindHandle(nvHandle);
721         if(s_cachedNvRef == 0)
722             return NULL;
723         NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
724         if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
725         {
726             s_cachedNvRamRef = NvRamGetIndex(nvHandle);
727             s_cachedNvIndex.publicArea.attributes =
728                 NvReadRamIndexAttributes(s_cachedNvRamRef);
729         }
730     }
731     if(locator != NULL)
732         *locator = s_cachedNvRef;
733     return &s_cachedNvIndex;
734 }
```

8.4.5.12 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

```
735  TPM_RC
736  NvWriteIndexData(
737      NV_INDEX        *nvIndex,       // IN: the description of the index
738      UINT32           offset,        // IN: offset of NV data
739      UINT32           size,          // IN: size of NV data
740      void            *data           // IN: data buffer
741      )
742  {
743      TPM_RC               result = TPM_RC_SUCCESS;
744  //
745      pAssert(nvIndex != NULL);
746      // Make sure that this is dealing with the 'default' index.
747      // Note: it is tempting to change the calling sequence so that the 'default' is
748      // presumed.
749      pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
750
751      // Validate that write falls within range of the index
752      pAssert(offset <= nvIndex->publicArea.dataSize
753              &&  size <= (nvIndex->publicArea.dataSize - offset));
754
755      // Update TPMA_NV_WRITTEN bit if necessary
756      if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
757      {
758          // Update the in memory version of the attributes
759          SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
760
761          // If this is not orderly, then update the NV version of
762          // the attributes
763          if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
764          {
765              result = NvWriteNvIndexAttributes(s_cachedNvRef,
766                                                nvIndex->publicArea.attributes);
767              if(result != TPM_RC_SUCCESS)
768                  return result;
769              // If this is a partial write of an ordinary index, clear the whole
770              // index.
771              if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
772                 && (nvIndex->publicArea.dataSize > size))
773                  _plat__NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
774                                       nvIndex->publicArea.dataSize);
775          }
776          else
777          {
778              // This is orderly so update the RAM version
779              MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
780                         &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
781              // If setting WRITTEN for an orderly counter, make sure that the
782              // state saved version of the counter is saved
783              if(IsNvCounterIndex(nvIndex->publicArea.attributes))
784                  SET_NV_UPDATE(UT_ORDERLY);
785              // If setting the written attribute on an ordinary index, make sure that
786              // the data is all cleared out in case there is a partial write. This
787              // is only necessary for ordinary indexes because all of the other types
788              // are always written in total.
789              else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
790                  MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
```

```
                            0, nvIndex->publicArea.dataSize);
        }
    }
    // If this is orderly data, write it to RAM
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
    {
        // Note: if this is the first write to a counter, the code above will queue
        // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
        // process of doing that write, it will also write the initial counter value

        // Update RAM
        MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);

        // And indicate that the TPM is no longer orderly
        g_clearOrderly = TRUE;
    }
    else
    {
        // Offset into the index to the first byte of the data to be written to NV
        result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
                                      size, data);
    }
    return result;
}
```

8.4.5.13 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)

```
TPM_RC
NvWriteUINT64Data(
    NV_INDEX        *nvIndex,       // IN: the description of the index
    UINT64           intValue       // IN: the value to write
    )
{
    BYTE            bytes[8];
    UINT64_TO_BYTE_ARRAY(intValue, bytes);
//
    return NvWriteIndexData(nvIndex, 0, 8, &bytes);
}
```

8.4.5.14 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;
```

```
837 //
838     // Marshal public area
839     buffer = marshalBuffer;
840     dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);
841
842     // hash public area
843     digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
844     CryptDigestUpdate(&hashState, dataSize, marshalBuffer);
845
846     // Complete digest leaving room for the nameAlg
847     CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
848
849     // Include the nameAlg
850     UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
851     name->t.size = digestSize + 2;
852     return name;
853 }
```

8.4.5.15 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.

```
854 TPM2B_NAME *
855 NvGetNameByIndexHandle(
856     TPMI_RH_NV_INDEX     handle,         // IN: handle of the index
857     TPM2B_NAME          *name            // OUT: name of the index
858     )
859 {
860     NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
861 //
862     return NvGetIndexName(nvIndex, name);
863 }
```

8.4.5.16 NvDefineIndex()

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

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```
864 TPM_RC
865 NvDefineIndex(
866     TPMS_NV_PUBLIC  *publicArea,    // IN: A template for an area to create.
867     TPM2B_AUTH      *authValue      // IN: The initial authorization value
868     )
869 {
870     // The buffer to be written to NV memory
871     NV_INDEX        nvIndex;            // the index data
872     UINT16          entrySize;          // size of entry
873     TPM_RC          result;
874 //
875     entrySize = sizeof(NV_INDEX);
876
877     // only allocate data space for indexes that are going to be written to NV.
878     // Orderly indexes don't need space.
879     if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
880         entrySize += publicArea->dataSize;
```

```
881         // Check if we have enough space to create the NV Index
882         // In this implementation, the only resource limitation is the available NV
883         // space (and possibly RAM space.)  Other implementation may have other
884         // limitation on counter or on NV slots
885         if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
886             return TPM_RC_NV_SPACE;
887
888         // if the index to be defined is RAM backed, check RAM space availability
889         // as well
890         if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
891            &&  !NvRamTestSpaceIndex(publicArea->dataSize))
892             return TPM_RC_NV_SPACE;
893         // Copy input value to nvBuffer
894         nvIndex.publicArea = *publicArea;
895
896         // Copy the authValue
897         nvIndex.authValue = *authValue;
898
899         // Add index to NV memory
900         result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED,
901                        (BYTE *)&nvIndex);
902         if(result == TPM_RC_SUCCESS)
903         {
904         // If the data of NV Index is RAM backed, add the data area in RAM as well
905             if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
906                 NvAddRAM(publicArea);
907         }
908         return result;
909     }
```

8.4.5.17 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

```
910     TPM_RC
911     NvAddEvictObject(
912         TPMI_DH_OBJECT   evictHandle,   // IN: new evict handle
913         OBJECT          *object         // IN: object to be added
914         )
915     {
916         TPM_HANDLE       temp = object->evictHandle;
917         TPM_RC           result;
918     //
919         // Check if we have enough space to add the evict object
920         // An evict object needs 8 bytes in index table + sizeof OBJECT
921         // In this implementation, the only resource limitation is the available NV
922         // space.  Other implementation may have other limitation on evict object
923         // handle space
924         if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
925             return TPM_RC_NV_SPACE;
926
927         // Set evict attribute and handle
928         object->attributes.evict = SET;
929         object->evictHandle = evictHandle;
930
931         // Now put this in NV
932         result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
933
```

```
934         // Put things back the way they were
935         object->attributes.evict = CLEAR;
936         object->evictHandle = temp;
937 
938         return result;
939     }
```

8.4.5.18 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

```
940     TPM_RC
941     NvDeleteIndex(
942         NV_INDEX        *nvIndex,       // IN: an in RAM index descriptor
943         NV_REF           entityAddr     // IN: location in NV
944         )
945     {
946         TPM_RC           result;
947     //
948         if(nvIndex != NULL)
949         {
950             // Whenever a counter is deleted, make sure that the MaxCounter value is
951             // updated to reflect the value
952             if(IsNvCounterIndex(nvIndex->publicArea.attributes)
953                && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
954                 NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
955             result = NvDelete(entityAddr);
956             if(result != TPM_RC_SUCCESS)
957                 return result;
958             // If the NV Index is RAM backed, delete the RAM data as well
959             if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
960                 NvDeleteRAM(nvIndex->publicArea.nvIndex);
961             NvIndexCacheInit();
962         }
963         return TPM_RC_SUCCESS;
964     }
```

8.4.5.19 NvDeleteEvict()

This function will delete a NV evict object. Will return success if object deleted or if it does not exist

```
965     TPM_RC
966     NvDeleteEvict(
967         TPM_HANDLE       handle         // IN: handle of entity to be deleted
968         )
969     {
970         NV_REF      entityAddr = NvFindEvict(handle, NULL);     // pointer to entity
971         TPM_RC      result = TPM_RC_SUCCESS;
972     //
973         if(entityAddr != 0)
974             result = NvDelete(entityAddr);
975         return result;
976     }
```

8.4.5.20 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

```
TPM_RC
NvFlushHierarchy(
    TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flushed.
    )
{
    NV_REF           iter = NV_REF_INIT;
    NV_REF           currentAddr;
    TPM_HANDLE       entityHandle;
    TPM_RC           result = TPM_RC_SUCCESS;
//
    while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
    {
        if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
        {
            NV_INDEX        nvIndex;
//
            // If flush endorsement or platform hierarchy, no NV Index would be
            // flushed
            if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
                continue;
            // Get the index information
            NvReadNvIndexInfo(currentAddr, &nvIndex);

            // For storage hierarchy, flush OwnerCreated index
            if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV,
                             PLATFORMCREATE))
            {
                // Delete the index (including RAM for orderly)
                result = NvDeleteIndex(&nvIndex, currentAddr);
                if(result != TPM_RC_SUCCESS)
                    break;
                // Re-iterate from beginning after a delete
                iter = NV_REF_INIT;
            }
        }
        else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
        {
            OBJECT_ATTRIBUTES           attributes;
//
            NvRead(&attributes,
                   (UINT32)(currentAddr
                            + sizeof(TPM_HANDLE)
                            + offsetof(OBJECT, attributes)),
                   sizeof(OBJECT_ATTRIBUTES));
            // If the evict object belongs to the hierarchy to be flushed...
            if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
               || (hierarchy == TPM_RH_OWNER && attributes.spsHierarchy == SET)
               || (hierarchy == TPM_RH_ENDORSEMENT
                   &&  attributes.epsHierarchy == SET))
            {
                // ...then delete the evict object
                result = NvDelete(currentAddr);
                if(result != TPM_RC_SUCCESS)
                    break;
```

```
1031                    // Re-iterate from beginning after a delete
1032                    iter = NV_REF_INIT;
1033                }
1034            }
1035            else
1036            {
1037                FAIL(FATAL_ERROR_INTERNAL);
1038            }
1039        }
1040        return result;
1041    }
```

8.4.5.21 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

```
1042    TPM_RC
1043    NvSetGlobalLock(
1044        void
1045        )
1046    {
1047        NV_REF           iter = NV_REF_INIT;
1048        NV_RAM_REF       ramIter = NV_RAM_REF_INIT;
1049        NV_REF           currentAddr;
1050        NV_RAM_REF       currentRamAddr;
1051        TPM_RC           result = TPM_RC_SUCCESS;
1052    //
1053        // Check all normal indexes
1054        while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1055        {
1056            TPMA_NV         attributes = NvReadNvIndexAttributes(currentAddr);
1057    //
1058            // See if it should be locked
1059            if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1060               &&  IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1061            {
1062                SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1063                result = NvWriteNvIndexAttributes(currentAddr, attributes);
1064                if(result != TPM_RC_SUCCESS)
1065                    return result;
1066            }
1067        }
1068        // Now search all the orderly attributes
1069        while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1070        {
1071            // See if it should be locked
1072            TPMA_NV         attributes = NvReadRamIndexAttributes(currentRamAddr);
1073            if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1074            {
1075                SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1076                NvWriteRamIndexAttributes(currentRamAddr, attributes);
1077            }
1078        }
1079        return result;
1080    }
```

8.4.5.22 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX_CAP_HANDLES

```
1081    static void
1082    InsertSort(
1083        TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
1084        UINT32           count,         // IN: maximum count in the handle list
1085        TPM_HANDLE       entityHandle   // IN: handle to be inserted
1086        )
1087    {
1088        UINT32          i, j;
1089        UINT32          originalCount;
1090    //
1091        // For a corner case that the maximum count is 0, do nothing
1092        if(count == 0)
1093            return;
1094        // For empty list, add the handle at the beginning and return
1095        if(handleList->count == 0)
1096        {
1097            handleList->handle[0] = entityHandle;
1098            handleList->count++;
1099            return;
1100        }
1101        // Check if the maximum of the list has been reached
1102        originalCount = handleList->count;
1103        if(originalCount < count)
1104            handleList->count++;
1105        // Insert the handle to the list
1106        for(i = 0; i < originalCount; i++)
1107        {
1108            if(handleList->handle[i] > entityHandle)
1109            {
1110                for(j = handleList->count - 1; j > i; j--)
1111                {
1112                    handleList->handle[j] = handleList->handle[j - 1];
1113                }
1114                break;
1115            }
1116        }
1117        // If a slot was found, insert the handle in this position
1118        if(i < originalCount || handleList->count > originalCount)
1119            handleList->handle[i] = entityHandle;
1120        return;
1121    }
```

8.4.5.23 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

```
1122    TPMI_YES_NO
1123    NvCapGetPersistent(
1124        TPMI_DH_OBJECT   handle,        // IN: start handle
1125        UINT32           count,         // IN: maximum number of returned handles
```

```
1126        TPML_HANDLE     *handleList     // OUT: list of handle
1127        )
1128    {
1129        TPMI_YES_NO             more = NO;
1130        NV_REF                  iter = NV_REF_INIT;
1131        NV_REF                  currentAddr;
1132        TPM_HANDLE              entityHandle;
1133    //
1134        pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1135
1136        // Initialize output handle list
1137        handleList->count = 0;
1138
1139        // The maximum count of handles we may return is MAX_CAP_HANDLES
1140        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1141
1142        while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1143        {
1144            // Ignore persistent handles that have values less than the input handle
1145            if(entityHandle < handle)
1146                continue;
1147            // if the handles in the list have reached the requested count, and there
1148            // are still handles need to be inserted, indicate that there are more.
1149            if(handleList->count == count)
1150                more = YES;
1151            // A handle with a value larger than start handle is a candidate
1152            // for return. Insert sort it to the return list.  Insert sort algorithm
1153            // is chosen here for simplicity based on the assumption that the total
1154            // number of NV indexes is small.  For an implementation that may allow
1155            // large number of NV indexes, a more efficient sorting algorithm may be
1156            // used here.
1157            InsertSort(handleList, count, entityHandle);
1158        }
1159        return more;
1160    }
```

8.4.5.24 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

```
1161    TPMI_YES_NO
1162    NvCapGetIndex(
1163        TPMI_DH_OBJECT   handle,        // IN: start handle
1164        UINT32           count,         // IN: max number of returned handles
1165        TPML_HANDLE     *handleList     // OUT: list of handle
1166        )
1167    {
1168        TPMI_YES_NO             more = NO;
1169        NV_REF                  iter = NV_REF_INIT;
1170        NV_REF                  currentAddr;
1171        TPM_HANDLE              nvHandle;
1172    //
1173        pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1174
1175        // Initialize output handle list
1176        handleList->count = 0;
1177
```

```
1178        // The maximum count of handles we may return is MAX_CAP_HANDLES
1179        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1180
1181        while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1182        {
1183            // Ignore index handles that have values less than the 'handle'
1184            if(nvHandle < handle)
1185                continue;
1186            // if the count of handles in the list has reached the requested count,
1187            // and there are still handles to report, set more.
1188            if(handleList->count == count)
1189                more = YES;
1190            // A handle with a value larger than start handle is a candidate
1191            // for return. Insert sort it to the return list.  Insert sort algorithm
1192            // is chosen here for simplicity based on the assumption that the total
1193            // number of NV indexes is small.  For an implementation that may allow
1194            // large number of NV indexes, a more efficient sorting algorithm may be
1195            // used here.
1196            InsertSort(handleList, count, nvHandle);
1197        }
1198        return more;
1199    }
```

8.4.5.25 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
1200    UINT32
1201    NvCapGetIndexNumber(
1202        void
1203        )
1204    {
1205        UINT32           num = 0;
1206        NV_REF           iter = NV_REF_INIT;
1207    //
1208        while(NvNextIndex(NULL, &iter) != 0)
1209            num++;
1210        return num;
1211    }
```

8.4.5.26 NvCapGetPersistentNumber()

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

```
1212    UINT32
1213    NvCapGetPersistentNumber(
1214        void
1215        )
1216    {
1217        UINT32          num = 0;
1218        NV_REF         iter = NV_REF_INIT;
1219        TPM_HANDLE      handle;
1220    //
1221        while(NvNextEvict(&handle, &iter) != 0)
1222            num++;
1223        return num;
1224    }
```

8.4.5.27 NvCapGetPersistentAvail()

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

```
1225  UINT32
1226  NvCapGetPersistentAvail(
1227      void
1228      )
1229  {
1230      UINT32          availNVSpace;
1231      UINT32          counterNum = NvCapGetCounterNumber();
1232      UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1233  //
1234      // Get the available space in NV storage
1235      availNVSpace = NvGetFreeBytes();
1236
1237      if(counterNum < MIN_COUNTER_INDICES)
1238      {
1239          // Some space has to be reserved for counter objects.
1240          reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1241          if(reserved > availNVSpace)
1242              availNVSpace = 0;
1243          else
1244              availNVSpace -= reserved;
1245      }
1246      return availNVSpace / NV_EVICT_OBJECT_SIZE;
1247  }
```

8.4.5.28 NvCapGetCounterNumber()

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

```
1248  UINT32
1249  NvCapGetCounterNumber(
1250      void
1251      )
1252  {
1253      NV_REF           iter = NV_REF_INIT;
1254      NV_REF           currentAddr;
1255      UINT32           num = 0;
1256  //
1257      while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1258      {
1259          TPMA_NV             attributes = NvReadNvIndexAttributes(currentAddr);
1260          if(IsNvCounterIndex(attributes))
1261              num++;
1262      }
1263      return num;
1264  }
```

8.4.5.29 NvSetStartupAttributes()

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

```
1265  static TPMA_NV
1266  NvSetStartupAttributes(
1267      TPMA_NV         attributes,         // IN: attributes to change
1268      STARTUP_TYPE     type               // IN: start up type
1269      )
1270  {
1271      // Clear read lock
```

```
1272        CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);
1273
1274        // Will change a non counter index to the unwritten state if:
1275        // a) TPMA_NV_CLEAR_STCLEAR is SET
1276        // b) orderly and TPM Reset
1277        if(!IsNvCounterIndex(attributes))
1278        {
1279            if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
1280               || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1281                   && (type == SU_RESET)))
1282                CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);
1283        }
1284        // Unlock any index that is not written or that does not have
1285        // TPMA_NV_WRITEDEFINE SET.
1286        if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
1287           || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
1288            CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1289        return attributes;
1290    }
```

8.4.5.30 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.

```
1291    BOOL
1292    NvEntityStartup(
1293        STARTUP_TYPE     type           // IN: start up type
1294        )
1295    {
1296        NV_REF               iter = NV_REF_INIT;
1297        NV_RAM_REF           ramIter = NV_RAM_REF_INIT;
1298        NV_REF               currentAddr;        // offset points to the current entity
1299        NV_RAM_REF           currentRamAddr;
1300        TPM_HANDLE           nvHandle;
1301        TPMA_NV              attributes;
1302    //
1303        // Restore RAM index data
1304        NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
1305
1306        // Initialize the max NV counter value
1307        NvSetMaxCount(NvGetMaxCount());
1308
1309        // If recovering from state save, do nothing else
1310        if(type == SU_RESUME)
1311            return TRUE;
1312        // Iterate all the NV Index to clear the locks
1313        while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1314        {
1315            attributes = NvReadNvIndexAttributes(currentAddr);
1316
1317            // If this is an orderly index, defer processing until loop below
1318            if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
1319                continue;
1320            // Set the attributes appropriate for this startup type
1321            attributes = NvSetStartupAttributes(attributes, type);
1322            NvWriteNvIndexAttributes(currentAddr, attributes);
```

```
1323        }
1324        // Iterate all the orderly indexes to clear the locks and initialize counters
1325        while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1326        {
1327            attributes = NvReadRamIndexAttributes(currentRamAddr);
1328
1329            attributes = NvSetStartupAttributes(attributes, type);
1330
1331            // update attributes in RAM
1332            NvWriteRamIndexAttributes(currentRamAddr, attributes);
1333
1334            // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1335            if(IsNvCounterIndex(attributes)
1336               && (g_prevOrderlyState == SU_NONE_VALUE))
1337            {
1338                UINT64      counter;
1339    //
1340                // Read the counter value last saved to NV.
1341                counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1342
1343                // Set the lower bits of counter to 1's
1344                counter |= MAX_ORDERLY_COUNT;
1345
1346                // Write back to RAM
1347                // NOTE: Do not want to force a write to NV here. The counter value will
1348                // stay in RAM until the next shutdown or rollover.
1349                UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1350            }
1351        }
1352        return TRUE;
1353    }
```

8.4.5.31 NvCapGetCounterAvail()

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

```
1354    UINT32
1355    NvCapGetCounterAvail(
1356        void
1357        )
1358    {
1359        UINT32          availNVSpace;
1360        UINT32          availRAMSpace;
1361        UINT32          persistentNum = NvCapGetPersistentNumber();
1362        UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1363    //
1364        // Get the available space in NV storage
1365        availNVSpace = NvGetFreeBytes();
1366
1367        if(persistentNum < MIN_EVICT_OBJECTS)
1368        {
1369            // Some space has to be reserved for evict object. Adjust availNVSpace.
1370            reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1371            if(reserved > availNVSpace)
1372                availNVSpace = 0;
1373            else
1374                availNVSpace -= reserved;
1375        }
1376        // Compute the available space in RAM
1377        availRAMSpace = (int)(RAM_ORDERLY_END - NvRamGetEnd());
1378
1379        // Return the min of counter number in NV and in RAM
1380        if(availNVSpace / NV_INDEX_COUNTER_SIZE
```

```
1381            > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1382            return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1383        else
1384            return availNVSpace / NV_INDEX_COUNTER_SIZE;
1385    }
```

8.4.5.32 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.

```
1386    NV_REF
1387    NvFindHandle(
1388        TPM_HANDLE          handle
1389        )
1390    {
1391        NV_REF              addr;
1392        NV_REF              iter = NV_REF_INIT;
1393        TPM_HANDLE          nextHandle;
1394    //
1395        while((addr = NvNext(&iter, &nextHandle)) != 0)
1396        {
1397            if(nextHandle == handle)
1398                break;
1399        }
1400        return addr;
1401    }
```

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.

```
1402    UINT64
1403    NvReadMaxCount(
1404        void
1405        )
1406    {
1407        return s_maxCounter;
1408    }
```

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.

```
1409    void
1410    NvUpdateMaxCount(
1411        UINT64           count
1412        )
1413    {
1414        if(count > s_maxCounter)
1415            s_maxCounter = count;
1416    }
```

8.4.6.4 NvSetMaxCount()

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

```
1417    void
1418    NvSetMaxCount(
1419        UINT64          value
1420        )
1421    {
1422        s_maxCounter = value;
1423    }
```

8.4.6.5 NvGetMaxCount()

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

```
1424    UINT64
1425    NvGetMaxCount(
1426        void
1427        )
1428    {
1429        NV_REF               iter = NV_REF_INIT;
1430        NV_REF               currentAddr;
1431        UINT64               maxCount;
1432    //
1433        // Find the end of list marker and initialize the NV Max Counter value.
1434        while((currentAddr = NvNext(&iter, NULL )) != 0);
1435        // 'iter' should be pointing at the end of list marker so read in the current
1436        // value of the s_maxCounter.
1437        NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1438
1439        return maxCount;
1440    }
```

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

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

8.5.3 Functions

8.5.3.1 NvInitStatic()

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

```
3   static void
4   NvInitStatic(
5       void
6       )
7   {
8       // In some implementations, the end of NV is variable and is set at boot time.
9       // This value will be the same for each boot, but is not necessarily known
10      // at compile time.
11      s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
12      return;
```

```
}
```

8.5.3.2 NvCheckState()

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

This function is called at the beginning of ExecuteCommand() before any potential check of *g_NvStatus*.

```
void
NvCheckState(
    void
    )
{
    int     func_return;
//
    func_return = _plat__IsNvAvailable();
    if(func_return == 0)
        g_NvStatus = TPM_RC_SUCCESS;
    else if(func_return == 1)
        g_NvStatus = TPM_RC_NV_UNAVAILABLE;
    else
        g_NvStatus = TPM_RC_NV_RATE;
    return;
}
```

8.5.3.3 NvCommit

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

```
BOOL
NvCommit(
    void
    )
{
    return (_plat__NvCommit() == 0);
}
```

8.5.3.4 NvPowerOn()

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

Return Value	Meaning
TRUE(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  BOOL
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      if(nvOffset + size <= NV_MEMORY_SIZE)
97      {
98          // Set the flag that a NV write happened
99          SET_NV_UPDATE(UT_NV);
100         return _plat__NvMemoryWrite(nvOffset, size, inBuffer);
101     }
102     return FALSE;
103 }
```

8.5.3.8 NvUpdatePersistent()

This function is used to update a value in the PERSISTENT_DATA structure and commits the value to NV.

```
104 void
105 NvUpdatePersistent(
106     UINT32           offset,        // IN: location in PERMANENT_DATA to be updated
107     UINT32           size,          // IN: size of the value
108     void            *buffer         // IN: the new data
109     )
110 {
111     pAssert(offset + size <= sizeof(gp));
112     MemoryCopy(&gp + offset, buffer, size);
113     NvWrite(offset, size, buffer);
114 }
```

8.5.3.9 NvClearPersistent()

This function is used to clear a persistent data entry and commit it to NV

```
115 void
116 NvClearPersistent(
117     UINT32           offset,        // IN: the offset in the PERMANENT_DATA
118                                     //     structure to be cleared (zeroed)
119     UINT32           size           // IN: number of bytes to clear
120     )
121 {
122     pAssert(offset + size <= sizeof(gp));
123     MemorySet((&gp) + offset, 0, size);
124     NvWrite(offset, size, (&gp) + offset);
125 }
```

8.5.3.10 NvReadPersistent()

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

```
126 void
127 NvReadPersistent(
128     void
129     )
130 {
131     NvRead(&gp, NV_PERSISTENT_DATA, sizeof(gp));
132     return;
133 }
```


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.

```
BOOL
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 TRUE;
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

```
66  BOOL
67  ObjectIsSequence(
68      OBJECT          *object         // IN: handle to be checked
69      )
70  {
71      pAssert(object != NULL);
72      return (object->attributes.hmacSeq == SET
73              || object->attributes.hashSeq == SET
74              || object->attributes.eventSeq == SET);
75  }
```

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.

```
76  OBJECT*
77  HandleToObject(
78      TPMI_DH_OBJECT   handle          // IN: handle of the object
79      )
80  {
81      UINT32              index;
82  //
83      // Return NULL if the handle references a permanent handle because there is no
84      // associated OBJECT.
85      if(HandleGetType(handle) == TPM_HT_PERMANENT)
86          return NULL;
87      // In this implementation, the handle is determined by the slot occupied by the
88      // object.
89      index = handle - TRANSIENT_FIRST;
90      pAssert(index < MAX_LOADED_OBJECTS);
91      pAssert(s_objects[index].attributes.occupied);
92      return &s_objects[index];
93  }
```

8.6.3.8 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.

```
94  void
95  GetQualifiedName(
96      TPMI_DH_OBJECT   handle,         // IN: handle of the object
97      TPM2B_NAME      *qualifiedName   // OUT: qualified name of the object
98      )
99  {
100     OBJECT      *object;
101 //
102     switch(HandleGetType(handle))
103     {
104         case TPM_HT_PERMANENT:
```

```
105             qualifiedName->t.size = sizeof(TPM_HANDLE);
106             UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
107             break;
108         case TPM_HT_TRANSIENT:
109             object = HandleToObject(handle);
110             if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
111                 qualifiedName->t.size = 0;
112             else
113                 // Copy the name
114                 *qualifiedName = object->qualifiedName;
115             break;
116         default:
117             FAIL(FATAL_ERROR_INTERNAL);
118     }
119     return;
120 }
```

8.6.3.9 ObjectGetHierarchy()

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

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

8.6.3.10 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 while ObjectGetHierarchy() takes an pointer to an object.

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

```
143 TPMI_RH_HIERARCHY
144 GetHeriarchy(
145     TPMI_DH_OBJECT   handle          // IN :object handle
146     )
147 {
148     OBJECT          *object = HandleToObject(handle);
149 //
150     return ObjectGetHierarchy(object);
151 }
```

8.6.3.11 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

```
152  OBJECT *
153  FindEmptyObjectSlot(
154      TPMI_DH_OBJECT  *handle         // OUT: (optional)
155      )
156  {
157      UINT32               i;
158      OBJECT              *object;
159  //
160      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
161      {
162          object = &s_objects[i];
163          if(object->attributes.occupied == CLEAR)
164          {
165              if(handle)
166                  *handle = i + TRANSIENT_FIRST;
167              // Initialize the object attributes
168              MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
169              return object;
170          }
171      }
172      return NULL;
173  }
```

8.6.3.12 ObjectAllocateSlot()

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

```
174  OBJECT *
175  ObjectAllocateSlot(
176      TPMI_DH_OBJECT  *handle        // OUT: handle of allocated object
177      )
178  {
179      OBJECT          *object = FindEmptyObjectSlot(handle);
180  //
181      if(object != NULL)
182      {
183          // if found, mark as occupied
184          ObjectSetInUse(object);
185      }
186      return object;
187  }
```

8.6.3.13 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.

```
188  void
189  ObjectSetLoadedAttributes(
190      OBJECT          *object,        // IN: object attributes to finalize
191      TPM_HANDLE       parentHandle   // IN: the parent handle
```

```
192     )
193 {
194     OBJECT              *parent = HandleToObject(parentHandle);
195     TPMA_OBJECT          objectAttributes = object->publicArea.objectAttributes;
196 //
197     // Copy the stClear attribute from the public area. This could be overwritten
198     // if the parent has stClear SET
199     object->attributes.stClear =
200         IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);
201     // If parent handle is a permanent handle, it is a primary (unless it is NULL
202     if(parent == NULL)
203     {
204         object->attributes.primary = SET;
205         switch(parentHandle)
206         {
207             case TPM_RH_ENDORSEMENT:
208                 object->attributes.epsHierarchy = SET;
209                 break;
210             case TPM_RH_OWNER:
211                 object->attributes.spsHierarchy = SET;
212                 break;
213             case TPM_RH_PLATFORM:
214                 object->attributes.ppsHierarchy = SET;
215                 break;
216             default:
217                 // Treat the temporary attribute as a hierarchy
218                 object->attributes.temporary = SET;
219                 object->attributes.primary = CLEAR;
220                 break;
221         }
222     }
223     else
224     {
225         // is this a stClear object
226         object->attributes.stClear =
227             (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
228              || (parent->attributes.stClear == SET));
229         object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
230         object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
231         object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
232         // An object is temporary if its parent is temporary or if the object
233         // is external
234         object->attributes.temporary = parent->attributes.temporary
235             || object->attributes.external;
236     }
237     // If this is an external object, set the QN == name but don't SET other
238     // key properties ('parent' or 'derived')
239     if(object->attributes.external)
240         object->qualifiedName = object->name;
241     else
242     {
243         // check attributes for different types of parents
244         if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
245            && !object->attributes.publicOnly
246            && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
247            && object->publicArea.nameAlg != TPM_ALG_NULL)
248         {
249             // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
250             // Otherwise, it is a derivation parent.
251             if(object->publicArea.type == TPM_ALG_KEYEDHASH)
252                 object->attributes.derivation = SET;
253             else
254                 object->attributes.isParent = SET;
255         }
256         ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
257                              &object->name, &object->qualifiedName);
```

```
258         }
259         // Set slot occupied
260         ObjectSetInUse(object);
261         return;
262     }
```

8.6.3.14 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.

```
263     TPM_RC
264     ObjectLoad(
265         OBJECT          *object,        // IN: pointer to object slot
266                                         //     object
267         OBJECT          *parent,        // IN: (optional) the parent object
268         TPMT_PUBLIC     *publicArea,    // IN: public area to be installed in the object
269         TPMT_SENSITIVE  *sensitive,     // IN: (optional) sensitive area to be
270                                         //      installed in the object
271         TPM_RC           blamePublic,   // IN: parameter number to associate with the
272                                         //     publicArea errors
273         TPM_RC           blameSensitive,// IN: parameter number to associate with the
274                                         //     sensitive area errors
275         TPM2B_NAME      *name           // IN: (optional)
276     )
277     {
278         TPM_RC           result = TPM_RC_SUCCESS;
279     //
280     // Do validations of public area object descriptions
281         pAssert(publicArea != NULL);
282
283         // Is this public only or a no-name object?
284         if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
285         {
286             // Need to have schemes checked so that we do the right thing with the
287             // public key.
288             result = SchemeChecks(NULL, publicArea);
289         }
290         else
291         {
292             // For any sensitive area, make sure that the seedSize is no larger than the
293             // digest size of nameAlg
294             if(sensitive->seedValue.t.size > CryptHashGetDigestSize(publicArea->nameAlg))
295                 return TPM_RCS_KEY_SIZE + blameSensitive;
296             // Check attributes and schemes for consistency
297             result = PublicAttributesValidation(parent, publicArea);
298         }
299         if(result != TPM_RC_SUCCESS)
300             return RcSafeAddToResult(result, blamePublic);
301
302     // Sensitive area and binding checks
303
304         // On load, check nothing if the parent is fixedTPM. For all other cases, validate
305         // the keys.
306         if((parent == NULL)
307            || ((parent != NULL) && !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
308                                                  TPMA_OBJECT, fixedTPM)))
309         {
310             // Do the cryptographic key validation
311             result = CryptValidateKeys(publicArea, sensitive, blamePublic,
312                                        blameSensitive);
313             if(result != TPM_RC_SUCCESS)
```

```
314             return result;
315     }
316 #if ALG_RSA
317     // If this is an RSA key, then expand the private exponent.
318     // Note: ObjectLoad() is only called by TPM2_Import() if the parent is fixedTPM.
319     // For any key that does not have a fixedTPM parent, the exponent is computed
320     // whenever it is loaded
321     if((publicArea->type == TPM_ALG_RSA) && (sensitive != NULL))
322     {
323         result = CryptRsaLoadPrivateExponent(publicArea, sensitive);
324         if(result != TPM_RC_SUCCESS)
325             return result;
326     }
327 #endif // ALG_RSA
328     // See if there is an object to populate
329     if((result == TPM_RC_SUCCESS) && (object != NULL))
330     {
331         // Initialize public
332         object->publicArea = *publicArea;
333         // Copy sensitive if there is one
334         if(sensitive == NULL)
335             object->attributes.publicOnly = SET;
336         else
337             object->sensitive = *sensitive;
338         // Set the name, if one was provided
339         if(name != NULL)
340             object->name = *name;
341         else
342             object->name.t.size = 0;
343     }
344     return result;
345 }
```

8.6.3.15 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.

```
346 static HASH_OBJECT *
347 AllocateSequenceSlot(
348     TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
349     TPM2B_AUTH      *auth           // IN: the authValue for the slot
350     )
351 {
352     HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocateSlot(newHandle);
353 //
354     // Validate that the proper location of the hash state data relative to the
355     // object state data. It would be good if this could have been done at compile
356     // time but it can't so do it in something that can be removed after debug.
357     cAssert(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));
358
359     if(object != NULL)
360     {
361
362     // Set the common values that a sequence object shares with an ordinary object
363         // First, clear all attributes
364         MemorySet(&object->objectAttributes, 0, sizeof(TPMA_OBJECT));
365
366         // The type is TPM_ALG_NULL
367         object->type = TPM_ALG_NULL;
368
369         // This has no name algorithm and the name is the Empty Buffer
370         object->nameAlg = TPM_ALG_NULL;
371
```

```
372         // A sequence object is considered to be in the NULL hierarchy so it should
373         // be marked as temporary so that it can't be persisted
374         object->attributes.temporary = SET;
375
376         // A sequence object is DA exempt.
377         SET_ATTRIBUTE(object->objectAttributes, TPMA_OBJECT, noDA);
378
379         // Copy the authorization value
380         if(auth != NULL)
381             object->auth = *auth;
382         else
383             object->auth.t.size = 0;
384     }
385     return object;
386 }
387 #if CC_HMAC_Start || CC_MAC_Start
```

8.6.3.16 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

```
388 TPM_RC
389 ObjectCreateHMACSequence(
390     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
391     OBJECT           *keyObject,    // IN: the object containing the HMAC key
392     TPM2B_AUTH       *auth,         // IN: authValue
393     TPMI_DH_OBJECT   *newHandle     // OUT: HMAC sequence object handle
394     )
395 {
396     HASH_OBJECT          *hmacObject;
397 //
398     // Try to allocate a slot for new object
399     hmacObject = AllocateSequenceSlot(newHandle, auth);
400
401     if(hmacObject == NULL)
402         return TPM_RC_OBJECT_MEMORY;
403     // Set HMAC sequence bit
404     hmacObject->attributes.hmacSeq = SET;
405
406 #if !SMAC_IMPLEMENTED
407     if(CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
408                       keyObject->sensitive.sensitive.bits.b.size,
409                       keyObject->sensitive.sensitive.bits.b.buffer) == 0)
410 #else
411     if(CryptMacStart(&hmacObject->state.hmacState,
412                       &keyObject->publicArea.parameters,
413                       hashAlg, &keyObject->sensitive.sensitive.any.b) == 0)
414 #endif // SMAC_IMPLEMENTED
415         return TPM_RC_FAILURE;
416     return TPM_RC_SUCCESS;
417 }
418 #endif
```

8.6.3.17 ObjectCreateHashSequence()

This function creates a hash sequence object.

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

```
419  TPM_RC
420  ObjectCreateHashSequence(
421      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
422      TPM2B_AUTH      *auth,           // IN: authValue
423      TPMI_DH_OBJECT  *newHandle       // OUT: sequence object handle
424      )
425  {
426      HASH_OBJECT         *hashObject = AllocateSequenceSlot(newHandle, auth);
427  //
428      // See if slot allocated
429      if(hashObject == NULL)
430          return TPM_RC_OBJECT_MEMORY;
431      // Set hash sequence bit
432      hashObject->attributes.hashSeq = SET;
433
434      // Start hash for hash sequence
435      CryptHashStart(&hashObject->state.hashState[0], hashAlg);
436
437      return TPM_RC_SUCCESS;
438  }
```

8.6.3.18 ObjectCreateEventSequence()

This function creates an event sequence object.

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

```
439  TPM_RC
440  ObjectCreateEventSequence(
441      TPM2B_AUTH      *auth,           // IN: authValue
442      TPMI_DH_OBJECT  *newHandle       // OUT: sequence object handle
443      )
444  {
445      HASH_OBJECT         *hashObject = AllocateSequenceSlot(newHandle, auth);
446      UINT32               count;
447      TPM_ALG_ID           hash;
448  //
449      // See if slot allocated
450      if(hashObject == NULL)
451          return TPM_RC_OBJECT_MEMORY;
452      // Set the event sequence attribute
453      hashObject->attributes.eventSeq = SET;
454
455      // Initialize hash states for each implemented PCR algorithms
456      for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
457          CryptHashStart(&hashObject->state.hashState[count], hash);
458      return TPM_RC_SUCCESS;
459  }
```

8.6.3.19 ObjectTerminateEvent()

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

```
460  void
461  ObjectTerminateEvent(
462      void
```

```
463         )
464     {
465         HASH_OBJECT         *hashObject;
466         int                  count;
467         BYTE                 buffer[MAX_DIGEST_SIZE];
468     //
469         hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
470
471         // Don't assume that this is a proper sequence object
472         if(hashObject->attributes.eventSeq)
473         {
474             // If it is, close any open hash contexts. This is done in case
475             // the cryptographic implementation has some context values that need to be
476             // cleaned up (hygiene).
477             //
478             for(count = 0; CryptHashGetAlgByIndex(count) != TPM_ALG_NULL; count++)
479             {
480                 CryptHashEnd(&hashObject->state.hashState[count], 0, buffer);
481             }
482             // Flush sequence object
483             FlushObject(g_DRTMHandle);
484         }
485         g_DRTMHandle = TPM_RH_UNASSIGNED;
486     }
```

8.6.3.20 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

```
487     OBJECT *
488     ObjectContextLoad(
489         ANY_OBJECT_BUFFER   *object,        // IN: pointer to object structure in saved
490                                             //     context
491         TPMI_DH_OBJECT      *handle         // OUT: object handle
492         )
493     {
494         OBJECT      *newObject = ObjectAllocateSlot(handle);
495     //
496         // Try to allocate a slot for new object
497         if(newObject != NULL)
498         {
499             // Copy the first part of the object
500             MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
501             // See if this is a sequence object
502             if(ObjectIsSequence(newObject))
503             {
504                 // If this is a sequence object, import the data
505                 SequenceDataImport((HASH_OBJECT *)newObject,
506                                    (HASH_OBJECT_BUFFER *)object);
507             }
508             else
509             {
510                 // Copy input object data to internal structure
511                 MemoryCopy(newObject, object, sizeof(OBJECT));
512             }
513         }
514         return newObject;
515     }
```

8.6.3.21 FlushObject()

This function frees an object slot.

This function requires that the object is loaded.

```
516 void
517 FlushObject(
518     TPMI_DH_OBJECT   handle         // IN: handle to be freed
519     )
520 {
521     UINT32      index = handle - TRANSIENT_FIRST;
522 //
523     pAssert(index < MAX_LOADED_OBJECTS);
524     // Clear all the object attributes
525     MemorySet((BYTE*)&(s_objects[index].attributes),
526               0, sizeof(OBJECT_ATTRIBUTES));
527     return;
528 }
```

8.6.3.22 ObjectFlushHierarchy()

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

```
529 void
530 ObjectFlushHierarchy(
531     TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flush
532     )
533 {
534     UINT16          i;
535 //
536     // iterate object slots
537     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
538     {
539         if(s_objects[i].attributes.occupied)          // If found an occupied slot
540         {
541             switch(hierarchy)
542             {
543                 case TPM_RH_PLATFORM:
544                     if(s_objects[i].attributes.ppsHierarchy == SET)
545                         s_objects[i].attributes.occupied = FALSE;
546                     break;
547                 case TPM_RH_OWNER:
548                     if(s_objects[i].attributes.spsHierarchy == SET)
549                         s_objects[i].attributes.occupied = FALSE;
550                     break;
551                 case TPM_RH_ENDORSEMENT:
552                     if(s_objects[i].attributes.epsHierarchy == SET)
553                         s_objects[i].attributes.occupied = FALSE;
554                     break;
555                 default:
556                     FAIL(FATAL_ERROR_INTERNAL);
557                     break;
558             }
559         }
560     }
561
562     return;
563 }
```

8.6.3.23 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

```
564  TPM_RC
565  ObjectLoadEvict(
566      TPM_HANDLE      *handle,        // IN:OUT: evict object handle.  If success, it
567                                      // will be replace by the loaded object handle
568      COMMAND_INDEX    commandIndex   // IN: the command being processed
569      )
570  {
571      TPM_RC          result;
572      TPM_HANDLE      evictHandle = *handle;   // Save the evict handle
573      OBJECT          *object;
574  //
575      // If this is an index that references a persistent object created by
576      // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
577      if(*handle >= PLATFORM_PERSISTENT)
578      {
579          // belongs to platform
580          if(g_phEnable == CLEAR)
581              return TPM_RC_HANDLE;
582      }
583      // belongs to owner
584      else if(gc.shEnable == CLEAR)
585          return TPM_RC_HANDLE;
586      // Try to allocate a slot for an object
587      object = ObjectAllocateSlot(handle);
588      if(object == NULL)
589          return TPM_RC_OBJECT_MEMORY;
590      // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
591      // may be returned at this point. This will mark the slot as containing
592      // a transient object so that it will be flushed at the end of the
593      // command
594      result = NvGetEvictObject(evictHandle, object);
595
596      // Bail out if this failed
597      if(result != TPM_RC_SUCCESS)
598          return result;
599      // check the object to see if it is in the endorsement hierarchy
600      // if it is and this is not a TPM2_EvictControl() command, indicate
601      // that the hierarchy is disabled.
602      // If the associated hierarchy is disabled, make it look like the
603      // handle is not defined
604      if(ObjectGetHierarchy(object) == TPM_RH_ENDORSEMENT
605         && gc.ehEnable == CLEAR
606         && GetCommandCode(commandIndex) != TPM_CC_EvictControl)
607          return TPM_RC_HANDLE;
608
609      return result;
610  }
```

8.6.3.24 ObjectComputeName()

This does the name computation from a public area (can be marshaled or not).

```
611 TPM2B_NAME *
612 ObjectComputeName(
613     UINT32            size,          // IN: the size of the area to digest
614     BYTE             *publicArea,    // IN: the public area to digest
615     TPM_ALG_ID        nameAlg,       // IN: the hash algorithm to use
616     TPM2B_NAME       *name           // OUT: Computed name
617     )
618 {
619     // Hash the publicArea into the name buffer leaving room for the nameAlg
620     name->t.size = CryptHashBlock(nameAlg, size, publicArea,
621                                   sizeof(name->t.name) - 2,
622                                   &name->t.name[2]);
623     // set the nameAlg
624     UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
625     name->t.size += 2;
626     return name;
627 }
```

8.6.3.25 PublicMarshalAndComputeName()

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

```
628 TPM2B_NAME *
629 PublicMarshalAndComputeName(
630     TPMT_PUBLIC      *publicArea,    // IN: public area of an object
631     TPM2B_NAME       *name           // OUT: name of the object
632     )
633 {
634     // Will marshal a public area into a template. This is because the internal
635     // format for a TPM2B_PUBLIC is a structure and not a simple BYTE buffer.
636     TPM2B_TEMPLATE       marshaled;     // this is big enough to hold a
637                                         //  marshaled TPMT_PUBLIC
638     BYTE                *buffer = (BYTE *)&marshaled.t.buffer;
639 //
640     // if the nameAlg is NULL then there is no name.
641     if(publicArea->nameAlg == TPM_ALG_NULL)
642         name->t.size = 0;
643     else
644     {
645         // Marshal the public area into its canonical form
646         marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
647         // and compute the name
648         ObjectComputeName(marshaled.t.size, marshaled.t.buffer,
649                           publicArea->nameAlg, name);
650     }
651     return name;
652 }
```

8.6.3.26 ComputeQualifiedName()

This function computes the qualified name of an object.

```
653 void
654 ComputeQualifiedName(
655     TPM_HANDLE        parentHandle,  // IN: parent's handle
656     TPM_ALG_ID        nameAlg,       // IN: name hash
657     TPM2B_NAME       *name,          // IN: name of the object
658     TPM2B_NAME       *qualifiedName  // OUT: qualified name of the object
659     )
660 {
661     HASH_STATE        hashState;   // hash state
662     TPM2B_NAME        parentName;
663 //
```

```
664         if(parentHandle == TPM_RH_UNASSIGNED)
665         {
666             MemoryCopy2B(&qualifiedName->b, &name->b, sizeof(qualifiedName->t.name));
667             *qualifiedName = *name;
668         }
669         else
670         {
671             GetQualifiedName(parentHandle, &parentName);
672 
673             //      QN_A = hash_A (QN of parent || NAME_A)
674 
675             // Start hash
676             qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
677 
678             // Add parent's qualified name
679             CryptDigestUpdate2B(&hashState, &parentName.b);
680 
681             // Add self name
682             CryptDigestUpdate2B(&hashState, &name->b);
683 
684             // Complete hash leaving room for the name algorithm
685             CryptHashEnd(&hashState, qualifiedName->t.size,
686                          &qualifiedName->t.name[2]);
687             UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
688             qualifiedName->t.size += 2;
689         }
690         return;
691     }
```

8.6.3.27 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

```
692     BOOL
693     ObjectIsStorage(
694         TPMI_DH_OBJECT   handle          // IN: object handle
695         )
696     {
697         OBJECT           *object = HandleToObject(handle);
698         TPMT_PUBLIC      *publicArea = ((object != NULL) ? &object->publicArea : NULL);
699     //
700         return (publicArea != NULL
701                 && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
702                 && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
703                 && !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
704                 && (object->publicArea.type == ALG_RSA_VALUE
705                     || object->publicArea.type == ALG_ECC_VALUE));
706     }
```

8.6.3.28 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

```
707  TPMI_YES_NO
708  ObjectCapGetLoaded(
709      TPMI_DH_OBJECT   handle,        // IN: start handle
710      UINT32           count,         // IN: count of returned handles
711      TPML_HANDLE     *handleList     // OUT: list of handle
712      )
713  {
714      TPMI_YES_NO          more = NO;
715      UINT32               i;
716  //
717      pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
718
719      // Initialize output handle list
720      handleList->count = 0;
721
722      // The maximum count of handles we may return is MAX_CAP_HANDLES
723      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
724
725      // Iterate object slots to get loaded object handles
726      for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
727      {
728          if(s_objects[i].attributes.occupied == TRUE)
729          {
730              // A valid transient object can not be the copy of a persistent object
731              pAssert(s_objects[i].attributes.evict == CLEAR);
732
733              if(handleList->count < count)
734              {
735                  // If we have not filled up the return list, add this object
736                  // handle to it
737                  handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
738                  handleList->count++;
739              }
740              else
741              {
742                  // If the return list is full but we still have loaded object
743                  // available, report this and stop iterating
744                  more = YES;
745                  break;
746              }
747          }
748      }
749
750      return more;
751  }
```

8.6.3.29 ObjectCapGetTransientAvail()

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

```
752  UINT32
753  ObjectCapGetTransientAvail(
754      void
755      )
756  {
757      UINT32      i;
758      UINT32      num = 0;
```

```
759     //
760         // Iterate object slot to get the number of unoccupied slots
761         for(i = 0; i < MAX_LOADED_OBJECTS; i++)
762         {
763             if(s_objects[i].attributes.occupied == FALSE) num++;
764         }
765
766         return num;
767     }
```

8.6.3.30 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

```
768     TPMA_OBJECT
769     ObjectGetPublicAttributes(
770         TPM_HANDLE       handle
771         )
772     {
773         return HandleToObject(handle)->publicArea.objectAttributes;
774     }
775     OBJECT_ATTRIBUTES
776     ObjectGetProperties(
777         TPM_HANDLE       handle
778         )
779     {
780         return HandleToObject(handle)->attributes;
781     }
```

8.7 PCR.c

8.7.1 Introduction

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

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

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

8.7.2 Includes, Defines, and Data Definitions

```
1   #define PCR_C
2   #include "Tpm.h"
```

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

```
3   static const PCR_Attributes s_initAttributes[] =
4   {
5       // PCR 0 - 15, static RTM
6       {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
7       {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
8       {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
9       {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
10
11      {0, 0x0F, 0x1F},        // PCR 16, Debug
12      {0, 0x10, 0x1C},        // PCR 17, Locality 4
13      {0, 0x10, 0x1C},        // PCR 18, Locality 3
14      {0, 0x10, 0x0C},        // PCR 19, Locality 2
15      {0, 0x14, 0x0E},        // PCR 20, Locality 1
16      {0, 0x14, 0x04},        // PCR 21, Dynamic OS
17      {0, 0x14, 0x04},        // PCR 22, Dynamic OS
18      {0, 0x0F, 0x1F},        // PCR 23, Application specific
19      {0, 0x0F, 0x1F}         // PCR 24, testing policy
20  };
21
22  //** Functions
23
24  //*** PCRBelongsAuthGroup()
25  // This function indicates if a PCR belongs to a group that requires an authValue
26  // in order to modify the PCR.  If it does, 'groupIndex' is set to value of
27  // the group index.  This feature of PCR is decided by the platform specification.
```

Return Value	Meaning
TRUE(1)	PCR belongs to an authorization group
FALSE(0)	PCR belongs to an authorization group

```
28  BOOL
29  PCRBelongsAuthGroup(
30      TPMI_DH_PCR      handle,         // IN: handle of PCR
31      UINT32          *groupIndex      // OUT: group index if PCR belongs a
32                                       //      group that allows authValue.  If PCR
33                                       //      does not belong to an authorization
```

```
34                                      //      group, the value in this parameter is
35                                      //      invalid
36      )
37  {
38  #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
39      // Platform specification determines to which authorization group a PCR belongs
40      // (if any). In this implementation, we assume there is only
41      // one authorization group which contains PCR[20-22].  If the platform
42      // specification requires differently, the implementation should be changed
43      // accordingly
44      if(handle >= 20 && handle <= 22)
45      {
46          *groupIndex = 0;
47          return TRUE;
48      }
49
50  #endif
51      return FALSE;
52  }
```

8.7.2.1 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 to a policy group
FALSE(0)	PCR does not belong to a policy group

```
53  BOOL
54  PCRBelongsPolicyGroup(
55      TPMI_DH_PCR      handle,        // IN: handle of PCR
56      UINT32          *groupIndex     // OUT: group index if PCR belongs a group that
57                                      //      allows policy.  If PCR does not belong to
58                                      //      a policy group, the value in this
59                                      //      parameter is invalid
60      )
61  {
62  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
63      // Platform specification decides if a PCR belongs to a policy group and
64      // belongs to which group.  In this implementation, we assume there is only
65      // one policy group which contains PCR20-22.  If the platform specification
66      // requires differently, the implementation should be changed accordingly
67      if(handle >= 20 && handle <= 22)
68      {
69          *groupIndex = 0;
70          return TRUE;
71      }
72  #endif
73      return FALSE;
74  }
```

8.7.2.2 PCRBelongsTCBGroup()

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

Return Value	Meaning
TRUE(1)	PCR belongs to a TCB group
FALSE(0)	PCR does not belong to a TCB group

```
75  static BOOL
76  PCRBelongsTCBGroup(
77      TPMI_DH_PCR      handle          // IN: handle of PCR
78      )
79  {
80  #if ENABLE_PCR_NO_INCREMENT == YES
81      // Platform specification decides if a PCR belongs to a TCB group.  In this
82      // implementation, we assume PCR[20-22] belong to TCB group.  If the platform
83      // specification requires differently, the implementation should be
84      // changed accordingly
85      if(handle >= 20 && handle <= 22)
86          return TRUE;
87
88  #endif
89      return FALSE;
90  }
```

8.7.2.3 PCRPolicyIsAvailable()

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

Return Value	Meaning
TRUE(1)	the PCR may be authorized by policy
FALSE(0)	the PCR does not allow policy

```
91  BOOL
92  PCRPolicyIsAvailable(
93      TPMI_DH_PCR      handle          // IN: PCR handle
94      )
95  {
96      UINT32           groupIndex;
97
98      return PCRBelongsPolicyGroup(handle, &groupIndex);
99  }
```

8.7.2.4 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.

```
100 TPM2B_AUTH *
101 PCRGetAuthValue(
102     TPMI_DH_PCR      handle          // IN: PCR handle
103     )
104 {
105     UINT32       groupIndex;
106
107     if(PCRBelongsAuthGroup(handle, &groupIndex))
108     {
109         return &gc.pcrAuthValues.auth[groupIndex];
110     }
111     else
112     {
113         return NULL;
```

```
    }
}
```

8.7.2.5 PCRGetAuthPolicy()

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

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

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

8.7.2.6 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.

```
void
PCRSimStart(
    void
    )
{
    UINT32  i;
#if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
    for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
    {
        gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
        gp.pcrPolicies.policy[i].t.size = 0;
    }
#endif
#if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
    for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
    {
        gc.pcrAuthValues.auth[i].t.size = 0;
    }
#endif
    // We need to give an initial configuration on allocated PCR before
    // receiving any TPM2_PCR_Allocate command to change this configuration
    // When the simulation environment starts, we allocate all the PCRs
    for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
    gp.pcrAllocated.count++)
    {
        gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
            = CryptHashGetAlgByIndex(gp.pcrAllocated.count);

        gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
```

```
164             = PCR_SELECT_MAX;
165         for(i = 0; i < PCR_SELECT_MAX; i++)
166             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
167             = 0xFF;
168     }
169
170     // Store the initial configuration to NV
171     NV_SYNC_PERSISTENT(pcrPolicies);
172     NV_SYNC_PERSISTENT(pcrAllocated);
173
174     return;
175 }
```

8.7.2.7 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

```
176 static BYTE *
177 GetSavedPcrPointer(
178     TPM_ALG_ID       alg,           // IN: algorithm for bank
179     UINT32           pcrIndex       // IN: PCR index in PCR_SAVE
180     )
181 {
182     BYTE            *retVal;
183     switch(alg)
184     {
185 #if ALG_SHA1
186         case ALG_SHA1_VALUE:
187             retVal = gc.pcrSave.sha1[pcrIndex];
188             break;
189 #endif
190 #if ALG_SHA256
191         case ALG_SHA256_VALUE:
192             retVal = gc.pcrSave.sha256[pcrIndex];
193             break;
194 #endif
195 #if ALG_SHA384
196         case ALG_SHA384_VALUE:
197             retVal = gc.pcrSave.sha384[pcrIndex];
198             break;
199 #endif
200
201 #if ALG_SHA512
202         case ALG_SHA512_VALUE:
203             retVal = gc.pcrSave.sha512[pcrIndex];
204             break;
205 #endif
206 #if ALG_SM3_256
207         case ALG_SM3_256_VALUE:
208             retVal = gc.pcrSave.sm3_256[pcrIndex];
209             break;
210 #endif
211         default:
212             FAIL(FATAL_ERROR_INTERNAL);
213     }
214     return retVal;
215 }
```

8.7.2.8 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

```
216  BOOL
217  PcrIsAllocated(
218      UINT32           pcr,           // IN: The number of the PCR
219      TPMI_ALG_HASH    hashAlg        // IN: The PCR algorithm
220      )
221  {
222      UINT32               i;
223      BOOL                 allocated = FALSE;
224
225      if(pcr < IMPLEMENTATION_PCR)
226      {
227          for(i = 0; i < gp.pcrAllocated.count; i++)
228          {
229              if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
230              {
231                  if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8])
232                      & (1 << (pcr % 8))) != 0)
233                      allocated = TRUE;
234                  else
235                      allocated = FALSE;
236                  break;
237              }
238          }
239      }
240      return allocated;
241  }
```

8.7.2.9 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

```
242  static BYTE *
243  GetPcrPointer(
244      TPM_ALG_ID       alg,           // IN: algorithm for bank
245      UINT32           pcrNumber      // IN: PCR number
246      )
247  {
248      static BYTE     *pcr = NULL;
249
250      if(!PcrIsAllocated(pcrNumber, alg))
251          return NULL;
252
253      switch(alg)
254      {
255  #if ALG_SHA1
256          case ALG_SHA1_VALUE:
257              pcr = s_pcrs[pcrNumber].sha1Pcr;
258              break;
```

```
259     #endif
260     #if ALG_SHA256
261             case ALG_SHA256_VALUE:
262                 pcr = s_pcrs[pcrNumber].sha256Pcr;
263                 break;
264     #endif
265     #if ALG_SHA384
266             case ALG_SHA384_VALUE:
267                 pcr = s_pcrs[pcrNumber].sha384Pcr;
268                 break;
269     #endif
270     #if ALG_SHA512
271             case ALG_SHA512_VALUE:
272                 pcr = s_pcrs[pcrNumber].sha512Pcr;
273                 break;
274     #endif
275     #if ALG_SM3_256
276             case ALG_SM3_256_VALUE:
277                 pcr = s_pcrs[pcrNumber].sm3_256Pcr;
278                 break;
279     #endif
280             default:
281                 FAIL(FATAL_ERROR_INTERNAL);
282                 break;
283         }
284         return pcr;
285     }
```

8.7.2.10 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

```
286     static BOOL
287     IsPcrSelected(
288         UINT32                pcr,          // IN: The number of the PCR
289         TPMS_PCR_SELECTION  *selection      // IN: The selection structure
290         )
291     {
292         BOOL                  selected;
293         selected = (pcr < IMPLEMENTATION_PCR
294                     && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
295         return selected;
296     }
```

8.7.2.11 FilterPcr()

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

```
297     static void
298     FilterPcr(
299         TPMS_PCR_SELECTION  *selection      // IN: input PCR selection
300         )
301     {
302         UINT32     i;
303         TPMS_PCR_SELECTION      *allocated = NULL;
304
305         // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
```

```
306         for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
307             selection->pcrSelect[i] = 0;
308 
309         // Find the internal configuration for the bank
310         for(i = 0; i < gp.pcrAllocated.count; i++)
311         {
312             if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
313             {
314                 allocated = &gp.pcrAllocated.pcrSelections[i];
315                 break;
316             }
317         }
318 
319         for(i = 0; i < selection->sizeofSelect; i++)
320         {
321             if(allocated == NULL)
322             {
323                 // If the required bank does not exist, clear input selection
324                 selection->pcrSelect[i] = 0;
325             }
326             else
327                 selection->pcrSelect[i] &= allocated->pcrSelect[i];
328         }
329 
330         return;
331     }
```

8.7.2.12 PcrDrtm()

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

```
332     void
333     PcrDrtm(
334         const TPMI_DH_PCR        pcrHandle,     // IN: the index of the PCR to be
335                                                 //     modified
336         const TPMI_ALG_HASH      hash,          // IN: the bank identifier
337         const TPM2B_DIGEST      *digest         // IN: the digest to modify the PCR
338         )
339     {
340         BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);
341 
342         if(pcrData != NULL)
343         {
344             // Rest the PCR to zeros
345             MemorySet(pcrData, 0, digest->t.size);
346 
347             // if the TPM has not started, then set the PCR to 0...04 and then extend
348             if(!TPMIsStarted())
349             {
350                 pcrData[digest->t.size - 1] = 4;
351             }
352             // Now, extend the value
353             PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
354         }
355     }
```

8.7.2.13 PCR_ClearAuth()

This function is used to reset the PCR authorization values. It is called on TPM2_Startup(CLEAR) and TPM2_Clear().

```
356     void
357     PCR_ClearAuth(
```

```
    void
    )
{
#if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
    int         j;
    for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
    {
        gc.pcrAuthValues.auth[j].t.size = 0;
    }
#endif
}
```

8.7.2.14 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
BOOL
PCRStartup(
    STARTUP_TYPE     type,          // IN: startup type
    BYTE             locality       // IN: startup locality
    )
{
    UINT32              pcr, j;
    UINT32              saveIndex = 0;

    g_pcrReConfig = FALSE;

    // Don't test for SU_RESET because that should be the default when nothing
    // else is selected
    if(type != SU_RESUME && type != SU_RESTART)
    {
        // PCR generation counter is cleared at TPM_RESET
        gr.pcrCounter = 0;
    }

    // Initialize/Restore PCR values
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        // On resume, need to know if this PCR had its state saved or not
        UINT32      stateSaved;

        if(type == SU_RESUME
           &&  s_initAttributes[pcr].stateSave == SET)
        {
            stateSaved = 1;
        }
        else
        {
            stateSaved = 0;
            PCRChanged(pcr);
        }

        // If this is the H-CRTM PCR and we are not doing a resume and we
        // had an H-CRTM event, then we don't change this PCR
        if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
            continue;

        // Iterate each hash algorithm bank
        for(j = 0; j < gp.pcrAllocated.count; j++)
        {
            TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
            BYTE            *pcrData = GetPcrPointer(hash, pcr);
            UINT16           pcrSize = CryptHashGetDigestSize(hash);

```

```
417             if(pcrData != NULL)
418             {
419                 // if state was saved
420                 if(stateSaved == 1)
421                 {
422                     // Restore saved PCR value
423                     BYTE    *pcrSavedData;
424                     pcrSavedData = GetSavedPcrPointer(
425                         gp.pcrAllocated.pcrSelections[j].hash,
426                         saveIndex);
427                     if(pcrSavedData == NULL)
428                         return FALSE;
429                     MemoryCopy(pcrData, pcrSavedData, pcrSize);
430                 }
431                 else
432                     // PCR was not restored by state save
433                 {
434                     // If the reset locality of the PCR is 4, then
435                     // the reset value is all one's, otherwise it is
436                     // all zero.
437                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
438                         MemorySet(pcrData, 0xFF, pcrSize);
439                     else
440                     {
441                         MemorySet(pcrData, 0, pcrSize);
442                         if(pcr == HCRTM_PCR)
443                             pcrData[pcrSize - 1] = locality;
444                     }
445                 }
446             }
447         }
448         saveIndex += stateSaved;
449     }
450     // Reset authValues on TPM2_Startup(CLEAR)
451     if(type != SU_RESUME)
452         PCR_ClearAuth();
453     return TRUE;
454 }
```

8.7.2.15 PCRStateSave()

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

```
455 void
456 PCRStateSave(
457     TPM_SU            type           // IN: startup type
458     )
459 {
460     UINT32            pcr, j;
461     UINT32            saveIndex = 0;
462
463     // if state save CLEAR, nothing to be done.  Return here
464     if(type == TPM_SU_CLEAR)
465         return;
466
467     // Copy PCR values to the structure that should be saved to NV
468     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
469     {
470         UINT32  stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
471
472         // Iterate each hash algorithm bank
473         for(j = 0; j < gp.pcrAllocated.count; j++)
474         {
475             BYTE    *pcrData;
```

```
476             UINT32  pcrSize;
477
478             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
479
480             if(pcrData != NULL)
481             {
482                 pcrSize
483                     = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
484
485                 if(stateSaved == 1)
486                 {
487                     // Restore saved PCR value
488                     BYTE    *pcrSavedData;
489                     pcrSavedData
490                         = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
491                                              saveIndex);
492                     MemoryCopy(pcrSavedData, pcrData, pcrSize);
493                 }
494             }
495         }
496         saveIndex += stateSaved;
497     }
498
499     return;
500 }
```

8.7.2.16 PCRIsStateSaved()

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

Return Value	Meaning
TRUE(1)	PCR is state saved
FALSE(0)	PCR is not state saved

```
501 BOOL
502 PCRIsStateSaved(
503     TPMI_DH_PCR      handle          // IN: PCR handle to be extended
504     )
505 {
506     UINT32              pcr = handle - PCR_FIRST;
507
508     if(s_initAttributes[pcr].stateSave == SET)
509         return TRUE;
510     else
511         return FALSE;
512 }
```

8.7.2.17 PCRIsResetAllowed()

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

Return Value	Meaning
TRUE(1)	TPM2_PCR_Reset() is allowed
FALSE(0)	TPM2_PCR_Reset() is not allowed

```
513 BOOL
514 PCRIsResetAllowed(
```

```
515     TPMI_DH_PCR      handle          // IN: PCR handle to be extended
516     )
517 {
518     UINT8               commandLocality;
519     UINT8               localityBits = 1;
520     UINT32              pcr = handle - PCR_FIRST;
521
522     // Check for the locality
523     commandLocality = _plat__LocalityGet();
524
525 #ifdef DRTM_PCR
526     // For a TPM that does DRTM, Reset is not allowed at locality 4
527     if(commandLocality == 4)
528         return FALSE;
529 #endif
530
531     localityBits = localityBits << commandLocality;
532     if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
533         return FALSE;
534     else
535         return TRUE;
536 }
```

8.7.2.18 PCRChanged()

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

```
537 void
538 PCRChanged(
539     TPM_HANDLE       pcrHandle      // IN: the handle of the PCR that changed.
540     )
541 {
542     // For the reference implementation, the only change that does not cause
543     // increment is a change to a PCR in the TCB group.
544     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
545     {
546         gr.pcrCounter++;
547         if(gr.pcrCounter == 0)
548             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
549     }
550 }
```

8.7.2.19 PCRIsExtendAllowed()

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

Return Value	Meaning
TRUE(1)	extend is allowed
FALSE(0)	extend is not allowed

```
551 BOOL
552 PCRIsExtendAllowed(
553     TPMI_DH_PCR      handle          // IN: PCR handle to be extended
554     )
555 {
556     UINT8               commandLocality;
```

```
557         UINT8               localityBits = 1;
558         UINT32              pcr = handle - PCR_FIRST;
559
560         // Check for the locality
561         commandLocality = _plat__LocalityGet();
562         localityBits = localityBits << commandLocality;
563         if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
564             return FALSE;
565         else
566             return TRUE;
567     }
```

8.7.2.20 PCRExtend()

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

```
568     void
569     PCRExtend(
570         TPMI_DH_PCR      handle,         // IN: PCR handle to be extended
571         TPMI_ALG_HASH    hash,           // IN: hash algorithm of PCR
572         UINT32           size,           // IN: size of data to be extended
573         BYTE            *data            // IN: data to be extended
574         )
575     {
576         BYTE                *pcrData;
577         HASH_STATE           hashState;
578         UINT16               pcrSize;
579
580         pcrData = GetPcrPointer(hash, handle - PCR_FIRST);
581
582         // Extend PCR if it is allocated
583         if(pcrData != NULL)
584         {
585             pcrSize = CryptHashGetDigestSize(hash);
586             CryptHashStart(&hashState, hash);
587             CryptDigestUpdate(&hashState, pcrSize, pcrData);
588             CryptDigestUpdate(&hashState, size, data);
589             CryptHashEnd(&hashState, pcrSize, pcrData);
590
591             // PCR has changed so update the pcrCounter if necessary
592             PCRChanged(handle);
593         }
594
595         return;
596     }
```

8.7.2.21 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

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

```
597     void
598     PCRComputeCurrentDigest(
599         TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
600         TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
601                                             //     output)
602         TPM2B_DIGEST        *digest         // OUT: digest
603         )
604     {
605         HASH_STATE               hashState;
606         TPMS_PCR_SELECTION      *select;
607         BYTE                    *pcrData;   // will point to a digest
```

```
608         UINT32                  pcrSize;
609         UINT32                  pcr;
610         UINT32                  i;
611
612         // Initialize the hash
613         digest->t.size = CryptHashStart(&hashState, hashAlg);
614         pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
615
616         // Iterate through the list of PCR selection structures
617         for(i = 0; i < selection->count; i++)
618         {
619             // Point to the current selection
620             select = &selection->pcrSelections[i]; // Point to the current selection
621             FilterPcr(select);      // Clear out the bits for unimplemented PCR
622
623             // Need the size of each digest
624             pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);
625
626             // Iterate through the selection
627             for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
628             {
629                 if(IsPcrSelected(pcr, select))         // Is this PCR selected
630                 {
631                     // Get pointer to the digest data for the bank
632                     pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
633                     pAssert(pcrData != NULL);
634                     CryptDigestUpdate(&hashState, pcrSize, pcrData);  // add to digest
635                 }
636             }
637         }
638         // Complete hash stack
639         CryptHashEnd2B(&hashState, &digest->b);
640
641         return;
642     }
```

8.7.2.22 PCRRead()

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

```
643     void
644     PCRRead(
645         TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
646                                             //     output)
647         TPML_DIGEST         *digest,        // OUT: digest
648         UINT32              *pcrCounter     // OUT: the current value of PCR generation
649                                             //     number
650         )
651     {
652         TPMS_PCR_SELECTION      *select;
653         BYTE                    *pcrData;       // will point to a digest
654         UINT32                   pcr;
655         UINT32                   i;
656
657         digest->count = 0;
658
659         // Iterate through the list of PCR selection structures
660         for(i = 0; i < selection->count; i++)
661         {
662             // Point to the current selection
663             select = &selection->pcrSelections[i]; // Point to the current selection
664             FilterPcr(select);      // Clear out the bits for unimplemented PCR
665
```

```
666             // Iterate through the selection
667             for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
668             {
669                 if(IsPcrSelected(pcr, select))          // Is this PCR selected
670                 {
671                     // Check if number of digest exceed upper bound
672                     if(digest->count > 7)
673                     {
674                         // Clear rest of the current select bitmap
675                         while(pcr < IMPLEMENTATION_PCR
676                                   // do not round up!
677                               && (pcr / 8) < select->sizeofSelect)
678                         {
679                             // do not round up!
680                             select->pcrSelect[pcr / 8] &= (BYTE)~(1 << (pcr % 8));
681                             pcr++;
682                         }
683                         // Exit inner loop
684                         break;
685                     }
686                     // Need the size of each digest
687                     digest->digests[digest->count].t.size =
688                         CryptHashGetDigestSize(selection->pcrSelections[i].hash);
689 
690                     // Get pointer to the digest data for the bank
691                     pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
692                     pAssert(pcrData != NULL);
693                     // Add to the data to digest
694                     MemoryCopy(digest->digests[digest->count].t.buffer,
695                                pcrData,
696                                digest->digests[digest->count].t.size);
697                     digest->count++;
698                 }
699             }
700             // If we exit inner loop because we have exceed the output upper bound
701             if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
702             {
703                 // Clear rest of the selection
704                 while(i < selection->count)
705                 {
706                     MemorySet(selection->pcrSelections[i].pcrSelect, 0,
707                               selection->pcrSelections[i].sizeofSelect);
708                     i++;
709                 }
710                 // exit outer loop
711                 break;
712             }
713         }
714 
715         *pcrCounter = gr.pcrCounter;
716 
717         return;
718     }
```

8.7.2.23 PCRAllocate()

This function is used to change the PCR allocation.

Error Returns	Meaning
TPM_RC_NO_RESULT	allocate failed
TPM_RC_PCR	improper allocation

```
719  TPM_RC
720  PCRAllocate(
721      TPML_PCR_SELECTION  *allocate,      // IN: required allocation
722      UINT32              *maxPCR,        // OUT: Maximum number of PCR
723      UINT32              *sizeNeeded,    // OUT: required space
724      UINT32              *sizeAvailable  // OUT: available space
725      )
726  {
727      UINT32                  i, j, k;
728      TPML_PCR_SELECTION      newAllocate;
729      // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
730      BOOL                    pcrHcrtm = FALSE;
731      BOOL                    pcrDrtm = FALSE;
732
733      // Create the expected new PCR allocation based on the existing allocation
734      // and the new input:
735      //  1. if a PCR bank does not appear in the new allocation, the existing
736      //     allocation of this PCR bank will be preserved.
737      //  2. if a PCR bank appears multiple times in the new allocation, only the
738      //     last one will be in effect.
739      newAllocate = gp.pcrAllocated;
740      for(i = 0; i < allocate->count; i++)
741      {
742          for(j = 0; j < newAllocate.count; j++)
743          {
744              // If hash matches, the new allocation covers the old allocation
745              // for this particular bank.
746              // The assumption is the initial PCR allocation (from manufacture)
747              // has all the supported hash algorithms with an assigned bank
748              // (possibly empty).  So there must be a match for any new bank
749              // allocation from the input.
750              if(newAllocate.pcrSelections[j].hash ==
751                 allocate->pcrSelections[i].hash)
752              {
753                  newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
754                  break;
755              }
756          }
757          // The j loop must exit with a match.
758          pAssert(j < newAllocate.count);
759      }
760
761      // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
762      *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
763      if(*maxPCR > IMPLEMENTATION_PCR)
764          *maxPCR = IMPLEMENTATION_PCR;
765
766      // Compute required size for allocation
767      *sizeNeeded = 0;
768      for(i = 0; i < newAllocate.count; i++)
769      {
770          UINT32      digestSize
771              = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
772  #if defined(DRTM_PCR)
773          // Make sure that we end up with at least one DRTM PCR
774          pcrDrtm = pcrDrtm || TestBit(DRTM_PCR,
775                                       newAllocate.pcrSelections[i].pcrSelect,
776                                       newAllocate.pcrSelections[i].sizeofSelect);
777
```

```
778     #else   // if DRTM PCR is not required, indicate that the allocation is OK
779             pcrDrtm = TRUE;
780     #endif
781
782     #if defined(HCRTM_PCR)
783             // and one HCRTM PCR (since this is usually PCR 0...)
784             pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR,
785                                            newAllocate.pcrSelections[i].pcrSelect,
786                                            newAllocate.pcrSelections[i].sizeofSelect);
787     #else
788             pcrHcrtm = TRUE;
789     #endif
790             for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
791             {
792                 BYTE        mask = 1;
793                 for(k = 0; k < 8; k++)
794                 {
795                     if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
796                         *sizeNeeded += digestSize;
797                     mask = mask << 1;
798                 }
799             }
800         }
801
802         if(!pcrDrtm || !pcrHcrtm)
803             return TPM_RC_PCR;
804
805         // In this particular implementation, we always have enough space to
806         // allocate PCR.  Different implementation may return a sizeAvailable less
807         // than the sizeNeed.
808         *sizeAvailable = sizeof(s_pcrs);
809
810         // Save the required allocation to NV.  Note that after NV is written, the
811         // PCR allocation in NV is no longer consistent with the RAM data
812         // gp.pcrAllocated.  The NV version reflect the allocate after next
813         // TPM_RESET, while the RAM version reflects the current allocation
814         NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
815
816         return TPM_RC_SUCCESS;
817     }
```

8.7.2.24 PCRSetValue()

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

```
818     void
819     PCRSetValue(
820         TPM_HANDLE       handle,        // IN: the handle of the PCR to set
821         INT8             initialValue   // IN: the value to set
822         )
823     {
824         int              i;
825         UINT32           pcr = handle - PCR_FIRST;
826         TPMI_ALG_HASH    hash;
827         UINT16           digestSize;
828         BYTE            *pcrData;
829
830         // Iterate supported PCR bank algorithms to reset
831         for(i = 0; i < HASH_COUNT; i++)
832         {
833             hash = CryptHashGetAlgByIndex(i);
834             // Prevent runaway
835             if(hash == TPM_ALG_NULL)
```

```
836                 break;
837
838             // Get a pointer to the data
839             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
840
841             // If the PCR is allocated
842             if(pcrData != NULL)
843             {
844                 // And the size of the digest
845                 digestSize = CryptHashGetDigestSize(hash);
846
847                 // Set the LSO to the input value
848                 pcrData[digestSize - 1] = initialValue;
849
850                 // Sign extend
851                 if(initialValue >= 0)
852                     MemorySet(pcrData, 0, digestSize - 1);
853                 else
854                     MemorySet(pcrData, -1, digestSize - 1);
855             }
856         }
857     }
```

8.7.2.25 PCRResetDynamics

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

```
858     void
859     PCRResetDynamics(
860         void
861         )
862     {
863         UINT32              pcr, i;
864
865         // Initialize PCR values
866         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
867         {
868             // Iterate each hash algorithm bank
869             for(i = 0; i < gp.pcrAllocated.count; i++)
870             {
871                 BYTE    *pcrData;
872                 UINT32  pcrSize;
873
874                 pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
875
876                 if(pcrData != NULL)
877                 {
878                     pcrSize =
879                         CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
880
881                     // Reset PCR
882                     // Any PCR can be reset by locality 4 should be reset to 0
883                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
884                         MemorySet(pcrData, 0, pcrSize);
885                 }
886             }
887         }
888         return;
889     }
```

8.7.2.26 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

Return Value	Meaning
YES	if the return count is 0
NO	if the return count is not 0

```
890   TPMI_YES_NO
891   PCRCapGetAllocation(
892       UINT32               count,          // IN: count of return
893       TPML_PCR_SELECTION  *pcrSelection    // OUT: PCR allocation list
894       )
895   {
896       if(count == 0)
897       {
898           pcrSelection->count = 0;
899           return YES;
900       }
901       else
902       {
903           *pcrSelection = gp.pcrAllocated;
904           return NO;
905       }
906   }
```

8.7.2.27 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
907   static void
908   PCRSetSelectBit(
909       UINT32           pcr,            // IN: PCR number
910       BYTE            *bitmap          // OUT: bit map to be set
911       )
912   {
913       bitmap[pcr / 8] |= (1 << (pcr % 8));
914       return;
915   }
```

8.7.2.28 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

```
916   static BOOL
917   PCRGetProperty(
918       TPM_PT_PCR               property,
919       TPMS_TAGGED_PCR_SELECT  *select
920       )
921   {
922       UINT32              pcr;
923       UINT32              groupIndex;
924
925       select->tag = property;
926       // Always set the bitmap to be the size of all PCR
927       select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
928
929       // Initialize bitmap
```

```
930         MemorySet(select->pcrSelect, 0, select->sizeofSelect);
931
932         // Collecting properties
933         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
934         {
935             switch(property)
936             {
937                 case TPM_PT_PCR_SAVE:
938                     if(s_initAttributes[pcr].stateSave == SET)
939                         PCRSetSelectBit(pcr, select->pcrSelect);
940                     break;
941                 case TPM_PT_PCR_EXTEND_L0:
942                     if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
943                         PCRSetSelectBit(pcr, select->pcrSelect);
944                     break;
945                 case TPM_PT_PCR_RESET_L0:
946                     if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
947                         PCRSetSelectBit(pcr, select->pcrSelect);
948                     break;
949                 case TPM_PT_PCR_EXTEND_L1:
950                     if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
951                         PCRSetSelectBit(pcr, select->pcrSelect);
952                     break;
953                 case TPM_PT_PCR_RESET_L1:
954                     if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
955                         PCRSetSelectBit(pcr, select->pcrSelect);
956                     break;
957                 case TPM_PT_PCR_EXTEND_L2:
958                     if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
959                         PCRSetSelectBit(pcr, select->pcrSelect);
960                     break;
961                 case TPM_PT_PCR_RESET_L2:
962                     if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
963                         PCRSetSelectBit(pcr, select->pcrSelect);
964                     break;
965                 case TPM_PT_PCR_EXTEND_L3:
966                     if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
967                         PCRSetSelectBit(pcr, select->pcrSelect);
968                     break;
969                 case TPM_PT_PCR_RESET_L3:
970                     if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
971                         PCRSetSelectBit(pcr, select->pcrSelect);
972                     break;
973                 case TPM_PT_PCR_EXTEND_L4:
974                     if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
975                         PCRSetSelectBit(pcr, select->pcrSelect);
976                     break;
977                 case TPM_PT_PCR_RESET_L4:
978                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
979                         PCRSetSelectBit(pcr, select->pcrSelect);
980                     break;
981                 case TPM_PT_PCR_DRTM_RESET:
982                     // DRTM reset PCRs are the PCR reset by locality 4
983                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
984                         PCRSetSelectBit(pcr, select->pcrSelect);
985                     break;
986 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
987                 case TPM_PT_PCR_POLICY:
988                     if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
989                         PCRSetSelectBit(pcr, select->pcrSelect);
990                     break;
991 #endif
992 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
993                 case TPM_PT_PCR_AUTH:
994                     if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
995                         PCRSetSelectBit(pcr, select->pcrSelect);
```

```
                break;
#endif
#if ENABLE_PCR_NO_INCREMENT == YES
            case TPM_PT_PCR_NO_INCREMENT:
                if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
#endif
            default:
                // If property is not supported, stop scanning PCR attributes
                // and return.
                return FALSE;
                break;
        }
    }
    return TRUE;
}
```

8.7.2.29 PCRCapGetProperties()

This function returns a list of PCR properties starting at *property*.

Return Value	Meaning
YES	if no more property is available
NO	if there are more properties not reported

```
TPMI_YES_NO
PCRCapGetProperties(
    TPM_PT_PCR                  property,       // IN: the starting PCR property
    UINT32                      count,          // IN: count of returned properties
    TPML_TAGGED_PCR_PROPERTY    *select         // OUT: PCR select
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    // Initialize output property list
    select->count = 0;

    // The maximum count of properties we may return is MAX_PCR_PROPERTIES
    if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;

    // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
    // value would never be less than TPM_PT_PCR_FIRST
    cAssert(TPM_PT_PCR_FIRST == 0);

    // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
    // implemented on the TPM.
    for(i = property; i <= TPM_PT_PCR_LAST; i++)
    {
        if(select->count < count)
        {
            // If we have not filled up the return list, add more properties to it
            if(PCRGetProperty(i, &select->pcrProperty[select->count]))
                // only increment if the property is implemented
                select->count++;
        }
        else
        {
            // If the return list is full but we still have properties
            // available, report this and stop iterating.
            more = YES;
```

```
1049                break;
1050            }
1051        }
1052        return more;
1053    }
```

8.7.2.30 PCRCapGetHandles()

This function is used to get a list of handles of PCR, started from *handle*. If *handle* exceeds the maximum PCR handle range, an empty list will be returned and the return value will be NO.

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

```
1054    TPMI_YES_NO
1055    PCRCapGetHandles(
1056        TPMI_DH_PCR      handle,         // IN: start handle
1057        UINT32           count,          // IN: count of returned handles
1058        TPML_HANDLE     *handleList      // OUT: list of handle
1059        )
1060    {
1061        TPMI_YES_NO     more = NO;
1062        UINT32          i;
1063
1064        pAssert(HandleGetType(handle) == TPM_HT_PCR);
1065
1066        // Initialize output handle list
1067        handleList->count = 0;
1068
1069        // The maximum count of handles we may return is MAX_CAP_HANDLES
1070        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1071
1072        // Iterate PCR handle range
1073        for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1074        {
1075            if(handleList->count < count)
1076            {
1077                // If we have not filled up the return list, add this PCR
1078                // handle to it
1079                handleList->handle[handleList->count] = i + PCR_FIRST;
1080                handleList->count++;
1081            }
1082            else
1083            {
1084                // If the return list is full but we still have PCR handle
1085                // available, report this and stop iterating
1086                more = YES;
1087                break;
1088            }
1089        }
1090        return more;
1091    }
```

8.8 PP.c

8.8.1 Introduction

This file contains the functions that support the physical presence operations of the TPM.

8.8.2 Includes

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

8.8.3 Functions

8.8.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that always require confirmation with physical presence. The array is an array of bits that has a correspondence with the command code.

This command should only ever be executable in a manufacturing setting or in a simulation.

When set, these cannot be cleared.

```
void
PhysicalPresencePreInstall_Init(
    void
    )
{
    COMMAND_INDEX        commandIndex;
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0, sizeof(gp.ppList));

    // Any command that is PP_REQUIRED should be SET
    for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
    {
        if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
           &&  s_commandAttributes[commandIndex] & PP_REQUIRED)
            SET_BIT(commandIndex, gp.ppList);
    }
    // Write PP list to NV
    NV_SYNC_PERSISTENT(ppList);
    return;
}
```

8.8.3.2 PhysicalPresenceCommandSet()

This function is used to set the indicator that a command requires PP confirmation.

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

    // if the command isn't implemented, the do nothing
    if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
        return;

    // only set the bit if this is a command for which PP is allowed
    if(s_commandAttributes[commandIndex] & PP_COMMAND)
```

```
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 closest to but 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 because it has the lowest value.

```
static void
ContextIdSetOldest(
    void
    )
{
    CONTEXT_SLOT    lowBits;
    CONTEXT_SLOT    entry;
    CONTEXT_SLOT    smallest = ((CONTEXT_SLOT)~0);  // Set to the maximum possible
    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  BOOL
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 TRUE;
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: A 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_CONTEXTS	the 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.

```
721 UINT32
722 SessionCapGetActiveNumber(
723     void
724     )
725 {
726     UINT32              i;
727     UINT32              num = 0;
728
729     // Iterate the context array to find the number of non-zero slots
730     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
731     {
732         if(gr.contextArray[i] != 0) num++;
733     }
734
735     return num;
736 }
```

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.

```
737 UINT32
738 SessionCapGetActiveAvail(
739     void
740     )
741 {
742     UINT32              i;
743     UINT32              num = 0;
744
745     // Iterate the context array to find the number of zero slots
746     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
747     {
748         if(gr.contextArray[i] == 0) num++;
749     }
750
751     return num;
752 }
```

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 "PlatformClock.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  BOOL
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 TRUE;
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

```
#include "Tpm.h"
typedef struct
{
    TPM_ALG_ID          algID;
    TPMA_ALGORITHM      attributes;
} ALGORITHM;
static const ALGORITHM    s_algorithms[] =
{
// The entries in this table need to be in ascending order but the table doesn't
// need to be full (gaps are allowed). One day, a tool might exist to fill in the
// table from the TPM_ALG description
#if ALG_RSA
    {TPM_ALG_RSA,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
#endif
#if ALG_TDES
    {TPM_ALG_TDES,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
#endif
#if ALG_SHA1
    {TPM_ALG_SHA1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
#endif

    {TPM_ALG_HMAC,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},

#if ALG_AES
    {TPM_ALG_AES,           TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
#endif
#if ALG_MGF1
    {TPM_ALG_MGF1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
#endif

    {TPM_ALG_KEYEDHASH,     TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 1, 1, 0, 0)},

#if ALG_XOR
    {TPM_ALG_XOR,           TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
#endif

#if ALG_SHA256
    {TPM_ALG_SHA256,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
#endif
#if ALG_SHA384
    {TPM_ALG_SHA384,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
#endif
#if ALG_SHA512
    {TPM_ALG_SHA512,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
#endif
#if ALG_SM3_256
    {TPM_ALG_SM3_256,       TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
#endif
#if ALG_SM4
    {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
118     };
```

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

```
119     TPMI_YES_NO
120     AlgorithmCapGetImplemented(
121         TPM_ALG_ID                   algID,     // IN: the starting algorithm ID
122         UINT32                       count,     // IN: count of returned algorithms
123         TPML_ALG_PROPERTY           *algList    // OUT: algorithm list
124         )
125     {
126         TPMI_YES_NO     more = NO;
127         UINT32          i;
128         UINT32          algNum;
129
130         // initialize output algorithm list
131         algList->count = 0;
132
133         // The maximum count of algorithms we may return is MAX_CAP_ALGS.
134         if(count > MAX_CAP_ALGS)
135             count = MAX_CAP_ALGS;
136
137         // Compute how many algorithms are defined in s_algorithms array.
138         algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
139
140         // Scan the implemented algorithm list to see if there is a match to 'algID'.
141         for(i = 0; i < algNum; i++)
142         {
143             // If algID is less than the starting algorithm ID, skip it
144             if(s_algorithms[i].algID < algID)
145                 continue;
146             if(algList->count < count)
147             {
148                 // If we have not filled up the return list, add more algorithms
149                 // to it
150                 algList->algProperties[algList->count].alg = s_algorithms[i].algID;
151                 algList->algProperties[algList->count].algProperties =
152                     s_algorithms[i].attributes;
153                 algList->count++;
154             }
155             else
156             {
157                 // If the return list is full but we still have algorithms
158                 // available, report this and stop scanning.
159                 more = YES;
160                 break;
161             }
162         }
163
164         return more;
165     }
```

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 there 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

```
AUTH_ROLE
CommandAuthRole(
    COMMAND_INDEX    commandIndex,  // IN: command index
    UINT32           handleIndex    // IN: handle index (zero based)
    )
{
    if(0 == handleIndex)
    {
        // Any authorization role set?
        COMMAND_ATTRIBUTES  properties = s_commandAttributes[commandIndex];

        if(properties & HANDLE_1_USER)
            return AUTH_USER;
        if(properties & HANDLE_1_ADMIN)
            return AUTH_ADMIN;
        if(properties & HANDLE_1_DUP)
            return AUTH_DUP;
    }
    else if(1 == handleIndex)
    {
        if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
            return AUTH_USER;
    }
    return AUTH_NONE;
}
```

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

```
int
EncryptSize(
    COMMAND_INDEX    commandIndex   // IN: command index
    )
{
    return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
            (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);
}
```

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      )
```

```
{
#ifdef  WRITE_LOCK
    return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
#else
    if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
    {
        switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
        {
            case TPM_CC_NV_Write:
#if CC_NV_Increment
            case TPM_CC_NV_Increment:
#endif
#if CC_NV_SetBits
            case TPM_CC_NV_SetBits:
#endif
#if CC_NV_Extend
            case TPM_CC_NV_Extend:
#endif
#if CC_AC_Send
            case TPM_CC_AC_Send:
#endif
            // NV write lock counts as a write operation for authorization purposes.
            // We check to see if the NV is write locked before we do the
            // authorization. If it is locked, we fail the command early.
            case TPM_CC_NV_WriteLock:
                return TRUE;
            default:
                break;
        }
    }
    return FALSE;
#endif
}
```

9.3.3.12 IsReadOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by write-lock.

```
BOOL
IsReadOperation(
    COMMAND_INDEX    commandIndex   // IN: Command to check
    )
{
#ifdef  READ_LOCK
    return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
#else

    if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
    {
        switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
        {
            case TPM_CC_NV_Read:
            case TPM_CC_PolicyNV:
            case TPM_CC_NV_Certify:
            // NV read lock counts as a read operation for authorization purposes.
            // We check to see if the NV is read locked before we do the
            // authorization. If it is locked, we fail the command early.
            case TPM_CC_NV_ReadLock:
                return TRUE;
            default:
                break;
        }
    }
    return FALSE;
```

```
#endif
}
```

9.3.3.13 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in *commandCode*.

Return Value	Meaning
YES	more command attributes are available
NO	no more command attributes are available

```
TPMI_YES_NO
CommandCapGetCCList(
    TPM_CC            commandCode,   // IN: start command code
    UINT32            count,         // IN: maximum count for number of entries in
                                     //     'commandList'
    TPML_CCA         *commandList    // OUT: list of TPMA_CC
    )
{
    TPMI_YES_NO       more = NO;
    COMMAND_INDEX     commandIndex;

    // 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

```
402 BOOL
403 IsVendorCommand(
404     COMMAND_INDEX    commandIndex   // IN: command index to check
405     )
406 {
407     return (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V));
408 }
```

9.4 Entity.c

9.4.1 Description

The functions in this file are used for accessing properties for handles of various types. Functions in other files require handles of a specific type but the functions in this file allow use of any handle type.

9.4.2 Includes

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

9.4.3 Functions

9.4.3.1 EntityGetLoadStatus()

This function will check that all the handles access loaded entities.

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_Hx	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
TPM_RC
EntityGetLoadStatus(
    COMMAND         *command        // IN/OUT: command parsing structure
    )
{
    UINT32                i;
    TPM_RC                result = TPM_RC_SUCCESS;
//
    for(i = 0; i < command->handleNum; i++)
    {
        TPM_HANDLE      handle = command->handles[i];
        switch(HandleGetType(handle))
        {
            // For handles associated with hierarchies, the entity is present
            // only if the associated enable is SET.
            case TPM_HT_PERMANENT:
                switch(handle)
                {
                    case TPM_RH_OWNER:
                        if(!gc.shEnable)
                            result = TPM_RC_HIERARCHY;
                        break;

#ifdef  VENDOR_PERMANENT
                    case VENDOR_PERMANENT:
#endif
                    case TPM_RH_ENDORSEMENT:
                        if(!gc.ehEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
                    case TPM_RH_PLATFORM:
                        if(!g_phEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
```

```
                        // null handle, PW session handle and lockout
                        // handle are always available
                    case TPM_RH_NULL:
                    case TPM_RS_PW:
                        // Need to be careful for lockout. Lockout is always available
                        // for policy checks but not always available when authValue
                        // is being checked.
                    case TPM_RH_LOCKOUT:
                    // Rather than have #ifdefs all over the code,
                    // CASE_ACT_HANDLE is defined in ACT.h. It is 'case TPM_RH_ACT_x:'
                    // FOR_EACH_ACT(CASE_ACT_HANDLE) creates a simple
                    // case TPM_RH_ACT_x: // for each of the implemented ACT.
                    FOR_EACH_ACT(CASE_ACT_HANDLE)
                        break;
                    default:
                        // If the implementation has a manufacturer-specific value
                        // then test for it here. Since this implementation does
                        // not have any, this implementation returns the same failure
                        // that unmarshaling of a bad handle would produce.
                        if(((TPM_RH)handle >= TPM_RH_AUTH_00)
                           && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
                            // if the implementation has a manufacturer-specific value
                            result = TPM_RC_VALUE;
                        else
                            // The handle is in the range of reserved handles but is
                            // not implemented in this TPM.
                            result = TPM_RC_VALUE;
                        break;
                }
                break;
            case TPM_HT_TRANSIENT:
                // For a transient object, check if the handle is associated
                // with a loaded object.
                if(!IsObjectPresent(handle))
                    result = TPM_RC_REFERENCE_H0;
                break;
            case TPM_HT_PERSISTENT:
                // Persistent object
                // Copy the persistent object to RAM and replace the handle with the
                // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
                // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
                // ObjectLoadEvict()
                result = ObjectLoadEvict(&command->handles[i], command->index);
                break;
            case TPM_HT_HMAC_SESSION:
                // For an HMAC session, see if the session is loaded
                // and if the session in the session slot is actually
                // an HMAC session.
                if(SessionIsLoaded(handle))
                {
                    SESSION             *session;
                    session = SessionGet(handle);
                    // Check if the session is a HMAC session
                    if(session->attributes.isPolicy == SET)
                        result = TPM_RC_HANDLE;
                }
                else
                    result = TPM_RC_REFERENCE_H0;
                break;
            case TPM_HT_POLICY_SESSION:
                // For a policy session, see if the session is loaded
                // and if the session in the session slot is actually
                // a policy session.
                if(SessionIsLoaded(handle))
                {
                    SESSION             *session;
```

```
102                     session = SessionGet(handle);
103                     // Check if the session is a policy session
104                     if(session->attributes.isPolicy == CLEAR)
105                         result = TPM_RC_HANDLE;
106                 }
107                 else
108                     result = TPM_RC_REFERENCE_H0;
109                 break;
110             case TPM_HT_NV_INDEX:
111                 // For an NV Index, use the TPM-specific routine
112                 // to search the IN Index space.
113                 result = NvIndexIsAccessible(handle);
114                 break;
115             case TPM_HT_PCR:
116                 // Any PCR handle that is unmarshaled successfully referenced
117                 // a PCR that is defined.
118                 break;
119 #if CC_AC_Send
120             case TPM_HT_AC:
121                 // Use the TPM-specific routine to search for the AC
122                 result = AcIsAccessible(handle);
123                 break;
124 #endif
125             default:
126                 // Any other handle type is a defect in the unmarshaling code.
127                 FAIL(FATAL_ERROR_INTERNAL);
128                 break;
129         }
130         if(result != TPM_RC_SUCCESS)
131         {
132             if(result == TPM_RC_REFERENCE_H0)
133                 result = result + i;
134             else
135                 result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
136             break;
137         }
138     }
139     return result;
140 }
```

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

```
141 UINT16
142 EntityGetAuthValue(
143     TPMI_DH_ENTITY   handle,        // IN: handle of entity
144     TPM2B_AUTH      *auth           // OUT: authValue of the entity
145     )
146 {
147     TPM2B_AUTH      *pAuth = NULL;
148
149     auth->t.size = 0;
150
151     switch(HandleGetType(handle))
```

```
152     {
153         case TPM_HT_PERMANENT:
154         {
155             switch(handle)
156             {
157                 case TPM_RH_OWNER:
158                     // ownerAuth for TPM_RH_OWNER
159                     pAuth = &gp.ownerAuth;
160                     break;
161                 case TPM_RH_ENDORSEMENT:
162                     // endorsementAuth for TPM_RH_ENDORSEMENT
163                     pAuth = &gp.endorsementAuth;
164                     break;
165                 // The ACT use platformAuth for auth
166                 FOR_EACH_ACT(CASE_ACT_HANDLE)
167                 case TPM_RH_PLATFORM:
168                     // platformAuth for TPM_RH_PLATFORM
169                     pAuth = &gc.platformAuth;
170                     break;
171                 case TPM_RH_LOCKOUT:
172                     // lockoutAuth for TPM_RH_LOCKOUT
173                     pAuth = &gp.lockoutAuth;
174                     break;
175                 case TPM_RH_NULL:
176                     // nullAuth for TPM_RH_NULL. Return 0 directly here
177                     return 0;
178                     break;
179 #ifdef  VENDOR_PERMANENT
180                 case VENDOR_PERMANENT:
181                     // vendor authorization value
182                     pAauth = &g_platformUniqueDetails;
183 #endif
184                 default:
185                     // If any other permanent handle is present it is
186                     // a code defect.
187                     FAIL(FATAL_ERROR_INTERNAL);
188                     break;
189             }
190             break;
191         }
192         case TPM_HT_TRANSIENT:
193             // authValue for an object
194             // A persistent object would have been copied into RAM
195             // and would have an transient object handle here.
196         {
197             OBJECT          *object;
198
199             object = HandleToObject(handle);
200             // special handling if this is a sequence object
201             if(ObjectIsSequence(object))
202             {
203                 pAuth = &((HASH_OBJECT *)object)->auth;
204             }
205             else
206             {
207                 // Authorization is available only when the private portion of
208                 // the object is loaded.  The check should be made before
209                 // this function is called
210                 pAssert(object->attributes.publicOnly == CLEAR);
211                 pAuth = &object->sensitive.authValue;
212             }
213         }
214         break;
215         case TPM_HT_NV_INDEX:
216             // authValue for an NV index
217         {
```

```
218                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
219                 pAssert(nvIndex != NULL);
220                 pAuth = &nvIndex->authValue;
221             }
222             break;
223             case TPM_HT_PCR:
224                 // authValue for PCR
225                 pAuth = PCRGetAuthValue(handle);
226                 break;
227             default:
228                 // If any other handle type is present here, then there is a defect
229                 // in the unmarshaling code.
230                 FAIL(FATAL_ERROR_INTERNAL);
231                 break;
232         }
233         // Copy the authValue
234         MemoryCopy2B((TPM2B *)auth, (TPM2B *)pAuth, sizeof(auth->t.buffer));
235         MemoryRemoveTrailingZeros(auth);
236         return auth->t.size;
237     }
```

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.

```
238     TPMI_ALG_HASH
239     EntityGetAuthPolicy(
240         TPMI_DH_ENTITY   handle,        // IN: handle of entity
241         TPM2B_DIGEST    *authPolicy     // OUT: authPolicy of the entity
242         )
243     {
244         TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
245         authPolicy->t.size = 0;
246 
247         switch(HandleGetType(handle))
248         {
249             case TPM_HT_PERMANENT:
250                 switch(handle)
251                 {
252                     case TPM_RH_OWNER:
253                         // ownerPolicy for TPM_RH_OWNER
254                         *authPolicy = gp.ownerPolicy;
255                         hashAlg = gp.ownerAlg;
256                         break;
257                     case TPM_RH_ENDORSEMENT:
258                         // endorsementPolicy for TPM_RH_ENDORSEMENT
259                         *authPolicy = gp.endorsementPolicy;
260                         hashAlg = gp.endorsementAlg;
261                         break;
262                     case TPM_RH_PLATFORM:
263                         // platformPolicy for TPM_RH_PLATFORM
264                         *authPolicy = gc.platformPolicy;
265                         hashAlg = gc.platformAlg;
266                         break;
267                     case TPM_RH_LOCKOUT:
268                         // lockoutPolicy for TPM_RH_LOCKOUT
269                         *authPolicy = gp.lockoutPolicy;
```

```
                    hashAlg = gp.lockoutAlg;
                    break;
#define ACT_GET_POLICY(N)                                                   \
                case TPM_RH_ACT_##N:                                        \
                    *authPolicy = go.ACT_##N.authPolicy;                    \
                    hashAlg = go.ACT_##N.hashAlg;                           \
                    break;
                    // Get the policy for each implemented ACT
                    FOR_EACH_ACT(ACT_GET_POLICY)
                default:
                    hashAlg = TPM_ALG_ERROR;
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            // authPolicy for an object
        {
            OBJECT *object = HandleToObject(handle);
            *authPolicy = object->publicArea.authPolicy;
            hashAlg = object->publicArea.nameAlg;
        }
        break;
        case TPM_HT_NV_INDEX:
            // authPolicy for a NV index
        {
            NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
            pAssert(nvIndex != 0);
            *authPolicy = nvIndex->publicArea.authPolicy;
            hashAlg = nvIndex->publicArea.nameAlg;
        }
        break;
        case TPM_HT_PCR:
            // authPolicy for a PCR
            hashAlg = PCRGetAuthPolicy(handle, authPolicy);
            break;
        default:
            // If any other handle type is present it is a code defect.
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }
    return hashAlg;
}
```

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

```
TPM2B_NAME *
EntityGetName(
    TPMI_DH_ENTITY   handle,        // IN: handle of entity
    TPM2B_NAME      *name           // OUT: name of entity
    )
{
    switch(HandleGetType(handle))
    {
        case TPM_HT_TRANSIENT:
        {
            // Name for an object
            OBJECT      *object = HandleToObject(handle);
            // an object with no nameAlg has no name
            if(object->publicArea.nameAlg == TPM_ALG_NULL)
                name->b.size = 0;
            else
                *name = object->name;
```

```
329             break;
330         }
331         case TPM_HT_NV_INDEX:
332             // Name for a NV index
333             NvGetNameByIndexHandle(handle, name);
334             break;
335         default:
336             // For all other types, the handle is the Name
337             name->t.size = sizeof(TPM_HANDLE);
338             UINT32_TO_BYTE_ARRAY(handle, name->t.name);
339             break;
340     }
341     return name;
342 }
```

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.

```
343 TPMI_RH_HIERARCHY
344 EntityGetHierarchy(
345     TPMI_DH_ENTITY   handle         // IN :handle of entity
346     )
347 {
348     TPMI_RH_HIERARCHY       hierarchy = TPM_RH_NULL;
349 
350     switch(HandleGetType(handle))
351     {
352         case TPM_HT_PERMANENT:
353             // hierarchy for a permanent handle
354             switch(handle)
355             {
356                 case TPM_RH_PLATFORM:
357                 case TPM_RH_ENDORSEMENT:
358                 case TPM_RH_NULL:
359                     hierarchy = handle;
360                     break;
361                 // all other permanent handles are associated with the owner
362                 // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
363                 default:
364                     hierarchy = TPM_RH_OWNER;
365                     break;
366             }
367             break;
368         case TPM_HT_NV_INDEX:
369             // hierarchy for NV index
370         {
371             NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
372             pAssert(nvIndex != NULL);
373 
374             // If only the platform can delete the index, then it is
375             // considered to be in the platform hierarchy, otherwise it
376             // is in the owner hierarchy.
377             if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV,
378                             PLATFORMCREATE))
379                 hierarchy = TPM_RH_PLATFORM;
380             else
381                 hierarchy = TPM_RH_OWNER;
```

```
382             }
383             break;
384             case TPM_HT_TRANSIENT:
385                 // hierarchy for an object
386             {
387                 OBJECT          *object;
388                 object = HandleToObject(handle);
389                 if(object->attributes.ppsHierarchy)
390                 {
391                     hierarchy = TPM_RH_PLATFORM;
392                 }
393                 else if(object->attributes.epsHierarchy)
394                 {
395                     hierarchy = TPM_RH_ENDORSEMENT;
396                 }
397                 else if(object->attributes.spsHierarchy)
398                 {
399                     hierarchy = TPM_RH_OWNER;
400                 }
401             }
402             break;
403             case TPM_HT_PCR:
404                 hierarchy = TPM_RH_OWNER;
405                 break;
406             default:
407                 FAIL(FATAL_ERROR_INTERNAL);
408                 break;
409         }
410         // this is unreachable but it provides a return value for the default
411         // case which makes the complier happy
412         return hierarchy;
413     }
```

9.5 Global.c

9.5.1 Description

This file will instance the TPM variables that are not stack allocated. Descriptions of global variables are in Global.h. There macro macro definitions that allows a variable to be instanced or simply defined as an external variable. When global.h is included from this .c file, GLOBAL_C is defined and values are instanced (and possibly initialized), but when global.h is included by any other file, they are simply defined as external values. DO NOT DEFINE GLOBAL_C IN ANY OTHER FILE.

NOTE: This is a change from previous implementations where Global.h just contained the extern declaration and values were instanced in this file. This change keeps the definition and instance in one file making maintenance easier. The instanced data will still be in the global.obj file.

The OIDs.h file works in a way that is similar to the Global.h with the definition of the values in OIDs.h such that they are instanced in global.obj. The macros that are defined in Global.h are used in OIDs.h in the same way as they are in Global.h.

9.5.2 Defines and Includes

```
1  #define GLOBAL_C
2  #include "Tpm.h"
3  #include "OIDs.h"
4  #if CC_CertifyX509
5  #   include "X509.h"
6  #endif // CC_CertifyX509
```

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
// Each of the implemented ACT
#define ACT_IMPLEMENTED_CASE(N)                                          \
            case TPM_RH_ACT_##N:

```

```
            FOR_EACH_ACT(ACT_IMPLEMENTED_CASE)

                return inHandle;
                break;
            default:
                break;
        }
    }
    // Out of range on the top
    return 0;
}
```

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

```
TPMI_YES_NO
PermanentCapGetHandles(
    TPM_HANDLE       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_PERMANENT);

    // 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 permanent handle range
    for(i = NextPermanentHandle(handle);
        i != 0; i = NextPermanentHandle(i + 1))
    {
        if(handleList->count < count)
        {
            // If we have not filled up the return list, add this permanent
            // handle to it
            handleList->handle[handleList->count] = i;
            handleList->count++;
        }
        else
        {
            // If the return list is full but we still have permanent handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```

9.6.3.4 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

```
TPMI_YES_NO
PermanentHandleGetPolicy(
    TPM_HANDLE           handle,        // IN: start handle
    UINT32               count,         // IN: max count of returned handles
    TPML_TAGGED_POLICY  *policyList     // OUT: list of handle
    )
{
    TPMI_YES_NO     more = NO;

    pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);

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

    // The maximum count of policies we may return is MAX_TAGGED_POLICIES
    if(count > MAX_TAGGED_POLICIES)
        count = MAX_TAGGED_POLICIES;

    // Iterate permanent handle range
    for(handle = NextPermanentHandle(handle);
        handle != 0;
        handle = NextPermanentHandle(handle + 1))
    {
        TPM2B_DIGEST    policyDigest;
        TPM_ALG_ID      policyAlg;
        // Check to see if this permanent handle has a policy
        policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
        if(policyAlg == TPM_ALG_ERROR)
           continue;
        if(policyList->count < count)
        {
            // If we have not filled up the return list, add this
            // policy to the list;
            policyList->policies[policyList->count].handle = handle;
            policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
            MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
                       policyDigest.t.buffer, policyDigest.t.size);
            policyList->count++;
        }
        else
        {
            // If the return list is full but we still have permanent handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```

9.7 IoBuffers.c

9.7.1 Includes and Data Definitions

This definition allows this module to **see** the values that are private to this module but kept in Global.c for ease of state migration.

```
#define IO_BUFFER_C
#include "Tpm.h"
#include "IoBuffers_fp.h"
```

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.

```
void
MemoryIoBufferAllocationReset(
    void
)
{
    s_actionIoAllocation = 0;
}
```

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.

```
void
MemoryIoBufferZero(
    void
)
{
    memset(s_actionIoBuffer, 0, s_actionIoAllocation);
}
```

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.

```
BYTE *
MemoryGetInBuffer(
    UINT32          size            // Size, in bytes, required for the input
                                    // unmarshaling
    )
{
    pAssert(size <= sizeof(s_actionIoBuffer));
    // In this implementation, a static buffer is set aside for the command action
    // buffers. The buffer is shared between input and output. This is because
    // 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

```
#define MANUFACTURE_C
#include "Tpm.h"
#include "TpmSizeChecks_fp.h"
```

9.9.3 Functions

9.9.3.1 TPM_Manufacture()

This function initializes the TPM values in preparation for the TPM's first use. This function will fail if previously called. The TPM can be re-manufactured by calling TPM_Teardown() first and then calling this function again.

Return Value	Meaning
-1	failure
0	success
1	manufacturing process previously performed

```
LIB_EXPORT int
TPM_Manufacture(
    int             firstTime       // IN: indicates if this is the first call from
                                    //     main()
    )
{
    TPM_SU          orderlyShutdown;

#if RUNTIME_SIZE_CHECKS
    // Call the function to verify the sizes of values that result from different
    // compile options.
    if(!TpmSizeChecks())
        return -1;
#endif
#if LIBRARY_COMPATIBILITY_CHECK
// Make sure that the attached library performs as expected.
    if(!MathLibraryCompatibilityCheck())
        return -1;
#endif

    // If TPM has been manufactured, return indication.
    if(!firstTime && g_manufactured)
        return 1;

    // Do power on initializations of the cryptographic libraries.
    CryptInit();

    s_DAPendingOnNV = FALSE;

    // initialize NV
    NvManufacture();
```

```
35
36      // Clear the magic value in the DRBG state
37      go.drbgState.magic = 0;
38
39      CryptStartup(SU_RESET);
40
41      // default configuration for PCR
42      PCRSimStart();
43
44      // initialize pre-installed hierarchy data
45      // This should happen after NV is initialized because hierarchy data is
46      // stored in NV.
47      HierarchyPreInstall_Init();
48
49      // initialize dictionary attack parameters
50      DAPreInstall_Init();
51
52      // initialize PP list
53      PhysicalPresencePreInstall_Init();
54
55      // initialize command audit list
56      CommandAuditPreInstall_Init();
57
58      // first start up is required to be Startup(CLEAR)
59      orderlyShutdown = TPM_SU_CLEAR;
60      NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
61
62      // initialize the firmware version
63      gp.firmwareV1 = FIRMWARE_V1;
64  #ifdef FIRMWARE_V2
65      gp.firmwareV2 = FIRMWARE_V2;
66  #else
67      gp.firmwareV2 = 0;
68  #endif
69      NV_SYNC_PERSISTENT(firmwareV1);
70      NV_SYNC_PERSISTENT(firmwareV2);
71
72      // initialize the total reset counter to 0
73      gp.totalResetCount = 0;
74      NV_SYNC_PERSISTENT(totalResetCount);
75
76      // initialize the clock stuff
77      go.clock = 0;
78      go.clockSafe = YES;
79
80      NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
81
82      // Commit NV writes.  Manufacture process is an artificial process existing
83      // only in simulator environment and it is not defined in the specification
84      // that what should be the expected behavior if the NV write fails at this
85      // point.  Therefore, it is assumed the NV write here is always success and
86      // no return code of this function is checked.
87      NvCommit();
88
89      g_manufactured = TRUE;
90
91      return 0;
92  }
```

9.9.3.2 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

```
 93  LIB_EXPORT int
 94  TPM_TearDown(
 95      void
 96      )
 97  {
 98      g_manufactured = FALSE;
 99      return 0;
100  }
```

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.

```
101  LIB_EXPORT void
102  TpmEndSimulation(
103      void
104      )
105  {
106  #if SIMULATION
107      HashLibSimulationEnd();
108      SymLibSimulationEnd();
109      MathLibSimulationEnd();
110  #if ALG_RSA
111      RsaSimulationEnd();
112  #endif
113  #if ALG_ECC
114      EccSimulationEnd();
115  #endif
116  #endif // SIMULATION
117  }
```

9.10 Marshal.c

9.10.1 Introduction

This file contains the marshaling and unmarshaling code.

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

9.10.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
1   TPM_RC
2   TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                            BOOL flag)
4   {
5       TPM_RC    result;
6       result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       if(*target == TPM_RH_NULL)
10      {
11          if(flag)
12              return TPM_RC_SUCCESS;
13          else
14              return TPM_RC_VALUE;
15      }
16      if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
17        &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
18              return TPM_RC_VALUE;
19      return TPM_RC_SUCCESS;
20  }
```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
1   UINT16
2   TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

```
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

An additional script is used to do the work that might be done by a linker or globally optimizing compiler. It searches for functions like TPMI_DH_OBJECT_Marshal() that do nothing but call another function and replaces the function with a #define.

```
#define TPMI_DH_OBJECT_Marshal(source, buffer, size)     \
      UINT32_Marshal((UINT32 *)source, buffer, size)
```

When replacing the function with a #define, the #define is placed in marshal_fp.h and the function body is removed from marshal.c.

9.10.3 Unmarshal and Marshal a Union

In TPM 2.0 Part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTE The Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

From this table, the following unmarshaling code is generated.

```
TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
                            UINT32 selector)
{
    switch(selector) {
#if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal(
                        (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
#endif
#if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal(
                    (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
#endif
#if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal(
                                (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
#endif
#if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal(
                                (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
#endif
    }
```

```
    return TPM_RC_SELECTOR;
}
```

NOTE The `#if/#endif` directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
UINT16
TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
                          UINT32 selector)
{
    switch(selector) {
#if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Marshal(
                        (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
#endif
#if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Marshal(
                        (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
#endif
#if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Marshal(
                         (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
#endif
#if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Marshal(
                          (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
#endif
    }
    assert(1);
    return 0;
}
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

9.10.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLIC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	“algorithm” associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
1   TPM_RC
2   TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
3   {
4       TPM_RC    result;
5       result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                       buffer, size, flag);
11      if(result != TPM_RC_SUCCESS)
12          return result;
13      result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                     buffer, size);
15      if(result != TPM_RC_SUCCESS)
16          return result;
17      result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                      buffer, size);
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                           buffer, size, (UINT32)target->type);
24      if(result != TPM_RC_SUCCESS)
25          return result;
26
27      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                        buffer, size, (UINT32)target->type);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31
32      return TPM_RC_SUCCESS;
33  }
```


The marshaling code for the TPMT_PUBLIC structure is:

```
UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
                                    (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
                                   (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
;
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                            (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));

    result = (UINT16)(result + TPM2B_DIGEST_Marshal(
                                 (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));

    result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
                            (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
                                                                   (UINT32)source->type));

    result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
                                   (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
                                                                   (UINT32)source->type));

    return result;
}
```

9.10.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
TPM_RC
TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    if( (target->count < 2))    // This check is triggered by the {2:} notation
                                // on 'count'
        return TPM_RC_SIZE;

    if((target->count) > 8)     // This check is triggered by the {:8} notation
```

```
                                    // on 'digests'.
        return TPM_RC_SIZE;

    result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
                                          buffer, size, (INT32)(target->count));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}

```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
UINT16
TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
                                                                        size));
    result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
                                    (TPM2B_DIGEST *)(source->digests), buffer, size,
                                    (INT32)(source->count)));

    return result;
}
```

The marshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```

9.10.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.10.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

When marshaling a TPM2B where the second member is a BYTE array, the size parameter indicates the size of the array. The second member can also be a structure. In this case, the caller does not prefill the size member. The marshaling code must marshal the structure and then back fill the calculated size.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
1   TPM_RC
2   TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8       // if size equal to 0, the rest of the structure is a zero buffer
9       //  so stop processing
10      if(target->t.size == 0)
11          return TPM_RC_SUCCESS;
12      if((target->t.size) > 1024)     // This check is triggered by the {:1024}
13                                      // notation on 'buffer'
14          return TPM_RC_SIZE;
15      result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
16                                    (INT32)(target->t.size));
17      if(result != TPM_RC_SUCCESS)
18          return result;
19      return TPM_RC_SUCCESS;
20  }
```

using these structure definitions:

```
1   typedef union {
2       struct {
3           UINT16        size;
4           BYTE          buffer[1024];
5       }             t;
6       TPM2B         b;
7   } TPM2B_EVENT;
```

9.10.7 Table Marshal Headers

9.10.7.1 TableMarshal.h

```
1   #ifndef _TABLE_DRIVEN_MARSHAL_H_
2   #define _TABLE_DRIVEN_MARSHAL_H_
```

These are the basic unmarshaling types. This is in the first byte of each structure descriptor that is passed to Marshal()/Unmarshal() for processing.

```
3   #define UINT_MTYPE          0
4   #define VALUES_MTYPE        (UINT_MTYPE + 1)
5   #define TABLE_MTYPE         (VALUES_MTYPE + 1)
6   #define MIN_MAX_MTYPE       (TABLE_MTYPE + 1)
7   #define ATTRIBUTES_MTYPE    (MIN_MAX_MTYPE + 1)
8   #define STRUCTURE_MTYPE     (ATTRIBUTES_MTYPE + 1)
9   #define TPM2B_MTYPE         (STRUCTURE_MTYPE + 1)
10  #define TPM2BS_MTYPE        (TPM2B_MTYPE + 1)
11  #define LIST_MTYPE          (TPM2BS_MTYPE + 1) // TPML
12  #define ERROR_MTYPE         (LIST_MTYPE + 1)
13  #define NULL_MTYPE          (ERROR_MTYPE + 1)
14  #define COMPOSITE_MTYPE     (NULL_MTYPE + 1)
```

9.10.7.1.1.1 The Marshal Index

A structure is used to hold the values that guide the marshaling/unmarshaling of each of the types. Each structure has a name and an address. For a structure to define a TPMS_name, the structure is a TPMS_name_MARSHAL_STRUCT and its index is TPMS_name_MARSHAL_INDEX. So, to get the proper structure, use the associated marshal index. The marshal index is passed to Marshal() or Unmarshal() and those functions look up the proper structure.

To handle structures that allow a null value, the upper bit of each marshal index indicates if the null value is allowed. This is the NULL_FLAG. It is defined in TableMarshalIndex.h because it is needed by code outside of the marshaling code. A structure will have a list of marshal indexes to indicate what to unmarshal. When that index appears in a structure/union, the value will contain a flag to indicate that the NULL_FLAG should be SET on the call to Unmarshal() to unmarshal the type. The caller simply takes the entry and passes it to Unmarshal() to indicate that the NULL_FLAG is SET. There is also the opportunity to SET the NULL_FLAG in the called structure if the NULL_FLAG was set in the call to the calling structure. This is indicated by:

```
15   #define NULL_MASK       ~(NULL_FLAG)
```

When looking up the value to marshal, the upper two bits of the marshal index are masked to yield the actual index.

```
16   typedef unsigned int        uint;
```

9.10.7.1.1.2 Modifier Octet Values

These are in used in anything that is an integer value. Theses would not be in structure modifier bytes (they would be used in values in structures but not the STRUCTURE_MTYPE header.

```
17  #define ONE_BYTES           (0)
18  #define TWO_BYTES           (1)
19  #define FOUR_BYTES          (2)
20  #define EIGHT_BYTES         (3)
21  #define SIZE_MASK           (0x3)
22  #define IS_SIGNED           (1 << 2)    // when the unmarshaled type is a signed value
23  #define SIGNED_MASK         (SIZE_MASK | IS_SIGNED)
```

This may be used for any type except a UINT_MTYPE

```
24  #define TAKES_NULL          (1 << 7)    // when the type takes a null
```

When referencing a structure, this flag indicates if a null is to be propagated to the referenced structure or type.

```
25  #define PROPAGATE_SHIFT     7
26  #define PROPAGATE_NULL      (1 << PROPAGATE_SHIFT)
```

Can be used in min-max or table structures.

```
27  #define HAS_BITS            (1 << 6)    // when bit mask is present
```

In a union, we need to know if this is a union of constant arrays.

```
28  #define IS_ARRAY_UNION      (1 << 6)
```

In a TPM2BS_MTYPE

```
29  #define SIZE_EQUAL          (1 << 6)
```

Right now, there are two spare bits in the modifiers field. Within the descriptor word of each entry in a StructMarsh_mst(), there is a selector field to determine which of the sub-types the entry represents and a field that is used to reference another structure entry. This is a 6-bit field allowing a structure to have 64 entries. This should be more than enough as the structures are not that long. As of now, only 10-bits of the descriptor word leaving room for expansion. These are the values used in a STRUCTURE_MTYPE to identify the sub-type of the thing being processed

```
30  #define SIMPLE_STYPE                0
31  #define UNION_STYPE                 1
32  #define ARRAY_STYPE                 2
```

The code used GET_ to get the element type and the compiler uses SET_ to initialize the value. The element type is the three bits (2:0).

```
33  #define GET_ELEMENT_TYPE(val)       (val & 7)
34  #define SET_ELEMENT_TYPE(val)       (val & 7)
```

When an entry is an array or union, this references the structure entry that contains the dimension or selector value. The code then uses this number to look up the structure entry for that element to find out what it and where is it in memory. When this is not a reference, it is a simple type and it could be used as an array value or a union selector. When a simple value, this field contains the size of the associated value (ONE_BYTES, TWO_BYTES ...) The entry size/number is 6 bits (13:8).


```
35  #define GET_ELEMENT_NUMBER(val)     (((val) >> 8) & 0x3F)
36  #define SET_ELEMENT_NUMBER(val)     (((val) & 0x3F) << 8)
37  #define GET_ELEMENT_SIZE(val)       GET_ELEMENT_NUMBER(val)
38  #define SET_ELEMENT_SIZE(val)       SET_ELEMENT_NUMBER(val)
```

This determines if the null flag is propagated to this type. If generate, the NULL_FLAG is SET in the index value. This flag is one bit (7)

```
39  #define ELEMENT_PROPAGATE           (1 << PROPAGATE_SHIFT)
40  #define INDEX_MASK                  ((UINT16)NULL_MASK)
```

This is used in all bit-field checks. These are used when a value that is checked is conditional (dependent on the compilation). For example, if AES_128 is (NO), then the bit associated with AES_128 will be 0. In some cases, the bit value is found by checking that the input is within the range of the table, and then using the (val - min) value to index the bit. This would be used when verifying that a particular algorithm is implemented. In other cases, there is a bit for each value in a table. For example, if checking the key sizes, there is a list of possible key sizes allowed by the algorithm registry and a bit field to indicate if that key size is allowed in the implementation. The smallest bit field has 32-bits 32-bits because it is implemented as part of the *values* array of the structures that allow bit fields.

```
41  #define IS_BIT_SET32(bit, bits)                                          \
42                          ((((UINT32 *)bits)[bit >> 5] & (1 << (bit & 0x1F))) != 0)
```

For a COMPOSITE_MTYPE, the qualifiers byte has an element size and count.

```
43  #define SET_ELEMENT_COUNT(count)    ((count & 0x1F) << 3)
44  #define GET_ELEMENT_COUNT(val)      ((val  >> 3) & 0x1F)
45  #endif // _TABLE_DRIVEN_MARSHAL_H_
```

9.10.7.2 TableMarshalData.h

```
1   #ifndef _Table_Marshal_Data_
2   #define _Table_Marshal_Data_
```

The datatype descriptions for each type if needed in addition to the default types.

```
3   typedef const struct TPM_ECC_CURVE_mst {
4       UINT8       marshalType;
5       UINT8       modifiers;
6       UINT8       errorCode;
7       UINT32      values[4];
8   } TPM_ECC_CURVE_mst;
9   typedef const struct TPM_CLOCK_ADJUST_mst {
10      UINT8       marshalType;
11      UINT8       modifiers;
12      UINT8       errorCode;
13      UINT32      values[2];
14  } TPM_CLOCK_ADJUST_mst;
15  typedef const struct TPM_EO_mst {
16      UINT8       marshalType;
17      UINT8       modifiers;
18      UINT8       errorCode;
19      UINT32      values[2];
20  } TPM_EO_mst;
21  typedef const struct TPM_SU_mst {
22      UINT8         marshalType;
23      UINT8         modifiers;
24      UINT8         errorCode;
25      UINT8         entries;
26      UINT32        values[2];
27  } TPM_SU_mst;
```

```
typedef const struct TPM_SE_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          entries;
    UINT32         values[3];
} TPM_SE_mst;
typedef const struct TPM_CAP_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          ranges;
    UINT8          singles;
    UINT32         values[3];
} TPM_CAP_mst;
typedef const struct TPMI_YES_NO_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          entries;
    UINT32         values[2];
} TPMI_YES_NO_mst;
typedef const struct TPMI_DH_OBJECT_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          ranges;
    UINT8          singles;
    UINT32         values[5];
} TPMI_DH_OBJECT_mst;
typedef const struct TPMI_DH_PARENT_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          ranges;
    UINT8          singles;
    UINT32         values[8];
} TPMI_DH_PARENT_mst;
typedef const struct TPMI_DH_PERSISTENT_mst {
    UINT8        marshalType;
    UINT8        modifiers;
    UINT8        errorCode;
    UINT32       values[2];
} TPMI_DH_PERSISTENT_mst;
typedef const struct TPMI_DH_ENTITY_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          ranges;
    UINT8          singles;
    UINT32         values[15];
} TPMI_DH_ENTITY_mst;
typedef const struct TPMI_DH_PCR_mst {
    UINT8        marshalType;
    UINT8        modifiers;
    UINT8        errorCode;
    UINT32       values[3];
} TPMI_DH_PCR_mst;
typedef const struct TPMI_SH_AUTH_SESSION_mst {
    UINT8          marshalType;
    UINT8          modifiers;
    UINT8          errorCode;
    UINT8          ranges;
    UINT8          singles;
    UINT32         values[5];
} TPMI_SH_AUTH_SESSION_mst;
```

```
94  typedef const struct TPMI_SH_HMAC_mst {
95      UINT8       marshalType;
96      UINT8       modifiers;
97      UINT8       errorCode;
98      UINT32      values[2];
99  } TPMI_SH_HMAC_mst;
100 typedef const struct TPMI_SH_POLICY_mst {
101     UINT8       marshalType;
102     UINT8       modifiers;
103     UINT8       errorCode;
104     UINT32      values[2];
105 } TPMI_SH_POLICY_mst;
106 typedef const struct TPMI_DH_CONTEXT_mst {
107     UINT8         marshalType;
108     UINT8         modifiers;
109     UINT8         errorCode;
110     UINT8         ranges;
111     UINT8         singles;
112     UINT32        values[6];
113 } TPMI_DH_CONTEXT_mst;
114 typedef const struct TPMI_DH_SAVED_mst {
115     UINT8         marshalType;
116     UINT8         modifiers;
117     UINT8         errorCode;
118     UINT8         ranges;
119     UINT8         singles;
120     UINT32        values[7];
121 } TPMI_DH_SAVED_mst;
122 typedef const struct TPMI_RH_HIERARCHY_mst {
123     UINT8         marshalType;
124     UINT8         modifiers;
125     UINT8         errorCode;
126     UINT8         entries;
127     UINT32        values[4];
128 } TPMI_RH_HIERARCHY_mst;
129 typedef const struct TPMI_RH_ENABLES_mst {
130     UINT8         marshalType;
131     UINT8         modifiers;
132     UINT8         errorCode;
133     UINT8         entries;
134     UINT32        values[5];
135 } TPMI_RH_ENABLES_mst;
136 typedef const struct TPMI_RH_HIERARCHY_AUTH_mst {
137     UINT8         marshalType;
138     UINT8         modifiers;
139     UINT8         errorCode;
140     UINT8         entries;
141     UINT32        values[4];
142 } TPMI_RH_HIERARCHY_AUTH_mst;
143 typedef const struct TPMI_RH_PLATFORM_mst {
144     UINT8         marshalType;
145     UINT8         modifiers;
146     UINT8         errorCode;
147     UINT8         entries;
148     UINT32        values[1];
149 } TPMI_RH_PLATFORM_mst;
150 typedef const struct TPMI_RH_OWNER_mst {
151     UINT8         marshalType;
152     UINT8         modifiers;
153     UINT8         errorCode;
154     UINT8         entries;
155     UINT32        values[2];
156 } TPMI_RH_OWNER_mst;
157 typedef const struct TPMI_RH_ENDORSEMENT_mst {
158     UINT8         marshalType;
159     UINT8         modifiers;
```

```
160         UINT8           errorCode;
161         UINT8           entries;
162         UINT32          values[2];
163     } TPMI_RH_ENDORSEMENT_mst;
164     typedef const struct TPMI_RH_PROVISION_mst {
165         UINT8           marshalType;
166         UINT8           modifiers;
167         UINT8           errorCode;
168         UINT8           entries;
169         UINT32          values[2];
170     } TPMI_RH_PROVISION_mst;
171     typedef const struct TPMI_RH_CLEAR_mst {
172         UINT8           marshalType;
173         UINT8           modifiers;
174         UINT8           errorCode;
175         UINT8           entries;
176         UINT32          values[2];
177     } TPMI_RH_CLEAR_mst;
178     typedef const struct TPMI_RH_NV_AUTH_mst {
179         UINT8           marshalType;
180         UINT8           modifiers;
181         UINT8           errorCode;
182         UINT8           ranges;
183         UINT8           singles;
184         UINT32          values[4];
185     } TPMI_RH_NV_AUTH_mst;
186     typedef const struct TPMI_RH_LOCKOUT_mst {
187         UINT8           marshalType;
188         UINT8           modifiers;
189         UINT8           errorCode;
190         UINT8           entries;
191         UINT32          values[1];
192     } TPMI_RH_LOCKOUT_mst;
193     typedef const struct TPMI_RH_NV_INDEX_mst {
194         UINT8         marshalType;
195         UINT8         modifiers;
196         UINT8         errorCode;
197         UINT32        values[2];
198     } TPMI_RH_NV_INDEX_mst;
199     typedef const struct TPMI_RH_AC_mst {
200         UINT8         marshalType;
201         UINT8         modifiers;
202         UINT8         errorCode;
203         UINT32        values[2];
204     } TPMI_RH_AC_mst;
205     typedef const struct TPMI_ALG_HASH_mst {
206         UINT8         marshalType;
207         UINT8         modifiers;
208         UINT8         errorCode;
209         UINT32        values[5];
210     } TPMI_ALG_HASH_mst;
211     typedef const struct TPMI_ALG_ASYM_mst {
212         UINT8         marshalType;
213         UINT8         modifiers;
214         UINT8         errorCode;
215         UINT32        values[5];
216     } TPMI_ALG_ASYM_mst;
217     typedef const struct TPMI_ALG_SYM_mst {
218         UINT8         marshalType;
219         UINT8         modifiers;
220         UINT8         errorCode;
221         UINT32        values[5];
222     } TPMI_ALG_SYM_mst;
223     typedef const struct TPMI_ALG_SYM_OBJECT_mst {
224         UINT8         marshalType;
225         UINT8         modifiers;
```

```
226         UINT8       errorCode;
227         UINT32      values[5];
228     } TPMI_ALG_SYM_OBJECT_mst;
229     typedef const struct TPMI_ALG_SYM_MODE_mst {
230         UINT8       marshalType;
231         UINT8       modifiers;
232         UINT8       errorCode;
233         UINT32      values[4];
234     } TPMI_ALG_SYM_MODE_mst;
235     typedef const struct TPMI_ALG_KDF_mst {
236         UINT8       marshalType;
237         UINT8       modifiers;
238         UINT8       errorCode;
239         UINT32      values[4];
240     } TPMI_ALG_KDF_mst;
241     typedef const struct TPMI_ALG_SIG_SCHEME_mst {
242         UINT8       marshalType;
243         UINT8       modifiers;
244         UINT8       errorCode;
245         UINT32      values[4];
246     } TPMI_ALG_SIG_SCHEME_mst;
247     typedef const struct TPMI_ECC_KEY_EXCHANGE_mst {
248         UINT8       marshalType;
249         UINT8       modifiers;
250         UINT8       errorCode;
251         UINT32      values[4];
252     } TPMI_ECC_KEY_EXCHANGE_mst;
253     typedef const struct TPMI_ST_COMMAND_TAG_mst {
254         UINT8         marshalType;
255         UINT8         modifiers;
256         UINT8         errorCode;
257         UINT8         entries;
258         UINT32        values[2];
259     } TPMI_ST_COMMAND_TAG_mst;
260     typedef const struct TPMI_ALG_MAC_SCHEME_mst {
261         UINT8       marshalType;
262         UINT8       modifiers;
263         UINT8       errorCode;
264         UINT32      values[5];
265     } TPMI_ALG_MAC_SCHEME_mst;
266     typedef const struct TPMI_ALG_CIPHER_MODE_mst {
267         UINT8       marshalType;
268         UINT8       modifiers;
269         UINT8       errorCode;
270         UINT32      values[4];
271     } TPMI_ALG_CIPHER_MODE_mst;
272     typedef const struct TPMS_EMPTY_mst
273     {
274         UINT8     marshalType;
275         UINT8     elements;
276         UINT16    values[3];
277     } TPMS_EMPTY_mst;
278     typedef const struct TPMS_ALGORITHM_DESCRIPTION_mst
279     {
280         UINT8     marshalType;
281         UINT8     elements;
282         UINT16    values[6];
283     } TPMS_ALGORITHM_DESCRIPTION_mst;
284     typedef struct TPMU_HA_mst
285     {
286         BYTE            countOfselectors;
287         BYTE            modifiers;
288         UINT16        offsetOfUnmarshalTypes;
289         UINT32        selectors[9];
290         UINT16        marshalingTypes[9];
291     } TPMU_HA_mst;
```

```
typedef const struct TPMT_HA_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[6];
} TPMT_HA_mst;
typedef const struct TPMS_PCR_SELECT_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[6];
} TPMS_PCR_SELECT_mst;
typedef const struct TPMS_PCR_SELECTION_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[9];
} TPMS_PCR_SELECTION_mst;
typedef const struct TPMT_TK_CREATION_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[9];
} TPMT_TK_CREATION_mst;
typedef const struct TPMT_TK_VERIFIED_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[9];
} TPMT_TK_VERIFIED_mst;
typedef const struct TPMT_TK_AUTH_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[9];
} TPMT_TK_AUTH_mst;
typedef const struct TPMT_TK_HASHCHECK_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[9];
} TPMT_TK_HASHCHECK_mst;
typedef const struct TPMS_ALG_PROPERTY_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[6];
} TPMS_ALG_PROPERTY_mst;
typedef const struct TPMS_TAGGED_PROPERTY_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[6];
} TPMS_TAGGED_PROPERTY_mst;
typedef const struct TPMS_TAGGED_PCR_SELECT_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[9];
} TPMS_TAGGED_PCR_SELECT_mst;
typedef const struct TPMS_TAGGED_POLICY_mst
{
    UINT8     marshalType;
    UINT8     elements;
    UINT16    values[6];
} TPMS_TAGGED_POLICY_mst;
```

```
358   typedef struct TPMU_CAPABILITIES_mst
359   {
360       BYTE            countOfselectors;
361       BYTE            modifiers;
362       UINT16        offsetOfUnmarshalTypes;
363       UINT32        selectors[10];
364       UINT16        marshalingTypes[10];
365   } TPMU_CAPABILITIES_mst;
366   typedef const struct TPMS_CAPABILITY_DATA_mst
367   {
368       UINT8     marshalType;
369       UINT8     elements;
370       UINT16    values[6];
371   } TPMS_CAPABILITY_DATA_mst;
372   typedef const struct TPMS_CLOCK_INFO_mst
373   {
374       UINT8     marshalType;
375       UINT8     elements;
376       UINT16    values[12];
377   } TPMS_CLOCK_INFO_mst;
378   typedef const struct TPMS_TIME_INFO_mst
379   {
380       UINT8     marshalType;
381       UINT8     elements;
382       UINT16    values[6];
383   } TPMS_TIME_INFO_mst;
384   typedef const struct TPMS_TIME_ATTEST_INFO_mst
385   {
386       UINT8     marshalType;
387       UINT8     elements;
388       UINT16    values[6];
389   } TPMS_TIME_ATTEST_INFO_mst;
390   typedef const struct TPMS_CERTIFY_INFO_mst
391   {
392       UINT8     marshalType;
393       UINT8     elements;
394       UINT16    values[6];
395   } TPMS_CERTIFY_INFO_mst;
396   typedef const struct TPMS_QUOTE_INFO_mst
397   {
398       UINT8     marshalType;
399       UINT8     elements;
400       UINT16    values[6];
401   } TPMS_QUOTE_INFO_mst;
402   typedef const struct TPMS_COMMAND_AUDIT_INFO_mst
403   {
404       UINT8     marshalType;
405       UINT8     elements;
406       UINT16    values[12];
407   } TPMS_COMMAND_AUDIT_INFO_mst;
408   typedef const struct TPMS_SESSION_AUDIT_INFO_mst
409   {
410       UINT8     marshalType;
411       UINT8     elements;
412       UINT16    values[6];
413   } TPMS_SESSION_AUDIT_INFO_mst;
414   typedef const struct TPMS_CREATION_INFO_mst
415   {
416       UINT8     marshalType;
417       UINT8     elements;
418       UINT16    values[6];
419   } TPMS_CREATION_INFO_mst;
420   typedef const struct TPMS_NV_CERTIFY_INFO_mst
421   {
422       UINT8     marshalType;
423       UINT8     elements;
```

```
424         UINT16    values[9];
425     } TPMS_NV_CERTIFY_INFO_mst;
426     typedef const struct TPMS_NV_DIGEST_CERTIFY_INFO_mst
427     {
428         UINT8     marshalType;
429         UINT8     elements;
430         UINT16    values[6];
431     } TPMS_NV_DIGEST_CERTIFY_INFO_mst;
432     typedef const struct TPMI_ST_ATTEST_mst {
433         UINT8         marshalType;
434         UINT8         modifiers;
435         UINT8         errorCode;
436         UINT8         ranges;
437         UINT8         singles;
438         UINT32        values[3];
439     } TPMI_ST_ATTEST_mst;
440     typedef struct TPMU_ATTEST_mst
441     {
442         BYTE            countOfselectors;
443         BYTE            modifiers;
444         UINT16        offsetOfUnmarshalTypes;
445         UINT32        selectors[8];
446         UINT16        marshalingTypes[8];
447     } TPMU_ATTEST_mst;
448     typedef const struct TPMS_ATTEST_mst
449     {
450         UINT8     marshalType;
451         UINT8     elements;
452         UINT16    values[21];
453     } TPMS_ATTEST_mst;
454     typedef const struct TPMS_AUTH_COMMAND_mst
455     {
456         UINT8     marshalType;
457         UINT8     elements;
458         UINT16    values[12];
459     } TPMS_AUTH_COMMAND_mst;
460     typedef const struct TPMS_AUTH_RESPONSE_mst
461     {
462         UINT8     marshalType;
463         UINT8     elements;
464         UINT16    values[9];
465     } TPMS_AUTH_RESPONSE_mst;
466     typedef const struct TPMI_TDES_KEY_BITS_mst {
467         UINT8         marshalType;
468         UINT8         modifiers;
469         UINT8         errorCode;
470         UINT8         entries;
471         UINT32        values[3];
472     } TPMI_TDES_KEY_BITS_mst;
473     typedef const struct TPMI_AES_KEY_BITS_mst {
474         UINT8         marshalType;
475         UINT8         modifiers;
476         UINT8         errorCode;
477         UINT8         entries;
478         UINT32        values[4];
479     } TPMI_AES_KEY_BITS_mst;
480     typedef const struct TPMI_SM4_KEY_BITS_mst {
481         UINT8         marshalType;
482         UINT8         modifiers;
483         UINT8         errorCode;
484         UINT8         entries;
485         UINT32        values[2];
486     } TPMI_SM4_KEY_BITS_mst;
487     typedef const struct TPMI_CAMELLIA_KEY_BITS_mst {
488         UINT8         marshalType;
489         UINT8         modifiers;
```

```
490         UINT8           errorCode;
491         UINT8           entries;
492         UINT32          values[4];
493     } TPMI_CAMELLIA_KEY_BITS_mst;
494     typedef struct TPMU_SYM_KEY_BITS_mst
495     {
496         BYTE              countOfselectors;
497         BYTE              modifiers;
498         UINT16          offsetOfUnmarshalTypes;
499         UINT32          selectors[6];
500         UINT16          marshalingTypes[6];
501     } TPMU_SYM_KEY_BITS_mst;
502     typedef struct TPMU_SYM_MODE_mst
503     {
504         BYTE              countOfselectors;
505         BYTE              modifiers;
506         UINT16          offsetOfUnmarshalTypes;
507         UINT32          selectors[6];
508         UINT16          marshalingTypes[6];
509     } TPMU_SYM_MODE_mst;
510     typedef const struct TPMT_SYM_DEF_mst
511     {
512         UINT8       marshalType;
513         UINT8       elements;
514         UINT16      values[9];
515     } TPMT_SYM_DEF_mst;
516     typedef const struct TPMT_SYM_DEF_OBJECT_mst
517     {
518         UINT8       marshalType;
519         UINT8       elements;
520         UINT16      values[9];
521     } TPMT_SYM_DEF_OBJECT_mst;
522     typedef const struct TPMS_SYMCIPHER_PARMS_mst
523     {
524         UINT8       marshalType;
525         UINT8       elements;
526         UINT16      values[3];
527     } TPMS_SYMCIPHER_PARMS_mst;
528     typedef const struct TPMS_DERIVE_mst
529     {
530         UINT8       marshalType;
531         UINT8       elements;
532         UINT16      values[6];
533     } TPMS_DERIVE_mst;
534     typedef const struct TPMS_SENSITIVE_CREATE_mst
535     {
536         UINT8       marshalType;
537         UINT8       elements;
538         UINT16      values[6];
539     } TPMS_SENSITIVE_CREATE_mst;
540     typedef const struct TPMS_SCHEME_HASH_mst
541     {
542         UINT8       marshalType;
543         UINT8       elements;
544         UINT16      values[3];
545     } TPMS_SCHEME_HASH_mst;
546     typedef const struct TPMS_SCHEME_ECDAA_mst
547     {
548         UINT8       marshalType;
549         UINT8       elements;
550         UINT16      values[6];
551     } TPMS_SCHEME_ECDAA_mst;
552     typedef const struct TPMI_ALG_KEYEDHASH_SCHEME_mst {
553         UINT8         marshalType;
554         UINT8         modifiers;
555         UINT8         errorCode;
```

```
556         UINT32      values[4];
557     } TPMI_ALG_KEYEDHASH_SCHEME_mst;
558     typedef const struct TPMS_SCHEME_XOR_mst
559     {
560         UINT8     marshalType;
561         UINT8     elements;
562         UINT16    values[6];
563     } TPMS_SCHEME_XOR_mst;
564     typedef struct TPMU_SCHEME_KEYEDHASH_mst
565     {
566         BYTE            countOfselectors;
567         BYTE            modifiers;
568         UINT16        offsetOfUnmarshalTypes;
569         UINT32        selectors[3];
570         UINT16        marshalingTypes[3];
571     } TPMU_SCHEME_KEYEDHASH_mst;
572     typedef const struct TPMT_KEYEDHASH_SCHEME_mst
573     {
574         UINT8     marshalType;
575         UINT8     elements;
576         UINT16    values[6];
577     } TPMT_KEYEDHASH_SCHEME_mst;
578     typedef struct TPMU_SIG_SCHEME_mst
579     {
580         BYTE            countOfselectors;
581         BYTE            modifiers;
582         UINT16        offsetOfUnmarshalTypes;
583         UINT32        selectors[8];
584         UINT16        marshalingTypes[8];
585     } TPMU_SIG_SCHEME_mst;
586     typedef const struct TPMT_SIG_SCHEME_mst
587     {
588         UINT8     marshalType;
589         UINT8     elements;
590         UINT16    values[6];
591     } TPMT_SIG_SCHEME_mst;
592     typedef struct TPMU_KDF_SCHEME_mst
593     {
594         BYTE            countOfselectors;
595         BYTE            modifiers;
596         UINT16        offsetOfUnmarshalTypes;
597         UINT32        selectors[5];
598         UINT16        marshalingTypes[5];
599     } TPMU_KDF_SCHEME_mst;
600     typedef const struct TPMT_KDF_SCHEME_mst
601     {
602         UINT8     marshalType;
603         UINT8     elements;
604         UINT16    values[6];
605     } TPMT_KDF_SCHEME_mst;
606     typedef const struct TPMI_ALG_ASYM_SCHEME_mst {
607         UINT8       marshalType;
608         UINT8       modifiers;
609         UINT8       errorCode;
610         UINT32      values[4];
611     } TPMI_ALG_ASYM_SCHEME_mst;
612     typedef struct TPMU_ASYM_SCHEME_mst
613     {
614         BYTE            countOfselectors;
615         BYTE            modifiers;
616         UINT16        offsetOfUnmarshalTypes;
617         UINT32        selectors[11];
618         UINT16        marshalingTypes[11];
619     } TPMU_ASYM_SCHEME_mst;
620     typedef const struct TPMT_ASYM_SCHEME_mst
621     {
```

```
622         UINT8     marshalType;
623         UINT8     elements;
624         UINT16    values[6];
625     } TPMT_ASYM_SCHEME_mst;
626     typedef const struct TPMI_ALG_RSA_SCHEME_mst {
627         UINT8       marshalType;
628         UINT8       modifiers;
629         UINT8       errorCode;
630         UINT32      values[4];
631     } TPMI_ALG_RSA_SCHEME_mst;
632     typedef const struct TPMT_RSA_SCHEME_mst
633     {
634         UINT8     marshalType;
635         UINT8     elements;
636         UINT16    values[6];
637     } TPMT_RSA_SCHEME_mst;
638     typedef const struct TPMI_ALG_RSA_DECRYPT_mst {
639         UINT8       marshalType;
640         UINT8       modifiers;
641         UINT8       errorCode;
642         UINT32      values[4];
643     } TPMI_ALG_RSA_DECRYPT_mst;
644     typedef const struct TPMT_RSA_DECRYPT_mst
645     {
646         UINT8     marshalType;
647         UINT8     elements;
648         UINT16    values[6];
649     } TPMT_RSA_DECRYPT_mst;
650     typedef const struct TPMI_RSA_KEY_BITS_mst {
651         UINT8         marshalType;
652         UINT8         modifiers;
653         UINT8         errorCode;
654         UINT8         entries;
655         UINT32        values[5];
656     } TPMI_RSA_KEY_BITS_mst;
657     typedef const struct TPMS_ECC_POINT_mst
658     {
659         UINT8     marshalType;
660         UINT8     elements;
661         UINT16    values[6];
662     } TPMS_ECC_POINT_mst;
663     typedef const struct TPMI_ALG_ECC_SCHEME_mst {
664         UINT8       marshalType;
665         UINT8       modifiers;
666         UINT8       errorCode;
667         UINT32      values[4];
668     } TPMI_ALG_ECC_SCHEME_mst;
669     typedef const struct TPMI_ECC_CURVE_mst {
670         UINT8       marshalType;
671         UINT8       modifiers;
672         UINT8       errorCode;
673         UINT32      values[3];
674     } TPMI_ECC_CURVE_mst;
675     typedef const struct TPMT_ECC_SCHEME_mst
676     {
677         UINT8     marshalType;
678         UINT8     elements;
679         UINT16    values[6];
680     } TPMT_ECC_SCHEME_mst;
681     typedef const struct TPMS_ALGORITHM_DETAIL_ECC_mst
682     {
683         UINT8     marshalType;
684         UINT8     elements;
685         UINT16    values[33];
686     } TPMS_ALGORITHM_DETAIL_ECC_mst;
687     typedef const struct TPMS_SIGNATURE_RSA_mst
```

```
688 {
689     UINT8     marshalType;
690     UINT8     elements;
691     UINT16    values[6];
692 } TPMS_SIGNATURE_RSA_mst;
693 typedef const struct TPMS_SIGNATURE_ECC_mst
694 {
695     UINT8     marshalType;
696     UINT8     elements;
697     UINT16    values[9];
698 } TPMS_SIGNATURE_ECC_mst;
699 typedef struct TPMU_SIGNATURE_mst
700 {
701     BYTE            countOfselectors;
702     BYTE            modifiers;
703     UINT16        offsetOfUnmarshalTypes;
704     UINT32        selectors[8];
705     UINT16        marshalingTypes[8];
706 } TPMU_SIGNATURE_mst;
707 typedef const struct TPMT_SIGNATURE_mst
708 {
709     UINT8     marshalType;
710     UINT8     elements;
711     UINT16    values[6];
712 } TPMT_SIGNATURE_mst;
713 typedef struct TPMU_ENCRYPTED_SECRET_mst
714 {
715     BYTE            countOfselectors;
716     BYTE            modifiers;
717     UINT16        offsetOfUnmarshalTypes;
718     UINT32        selectors[4];
719     UINT16        marshalingTypes[4];
720 } TPMU_ENCRYPTED_SECRET_mst;
721 typedef const struct TPMI_ALG_PUBLIC_mst {
722     UINT8       marshalType;
723     UINT8       modifiers;
724     UINT8       errorCode;
725     UINT32      values[4];
726 } TPMI_ALG_PUBLIC_mst;
727 typedef struct TPMU_PUBLIC_ID_mst
728 {
729     BYTE            countOfselectors;
730     BYTE            modifiers;
731     UINT16        offsetOfUnmarshalTypes;
732     UINT32        selectors[4];
733     UINT16        marshalingTypes[4];
734 } TPMU_PUBLIC_ID_mst;
735 typedef const struct TPMS_KEYEDHASH_PARMS_mst
736 {
737     UINT8     marshalType;
738     UINT8     elements;
739     UINT16    values[3];
740 } TPMS_KEYEDHASH_PARMS_mst;
741 typedef const struct TPMS_ASYM_PARMS_mst
742 {
743     UINT8     marshalType;
744     UINT8     elements;
745     UINT16    values[6];
746 } TPMS_ASYM_PARMS_mst;
747 typedef const struct TPMS_RSA_PARMS_mst
748 {
749     UINT8     marshalType;
750     UINT8     elements;
751     UINT16    values[12];
752 } TPMS_RSA_PARMS_mst;
753 typedef const struct TPMS_ECC_PARMS_mst
```

```
754   {
755       UINT8     marshalType;
756       UINT8     elements;
757       UINT16    values[12];
758   } TPMS_ECC_PARMS_mst;
759   typedef struct TPMU_PUBLIC_PARMS_mst
760   {
761       BYTE            countOfselectors;
762       BYTE            modifiers;
763       UINT16        offsetOfUnmarshalTypes;
764       UINT32        selectors[4];
765       UINT16        marshalingTypes[4];
766   } TPMU_PUBLIC_PARMS_mst;
767   typedef const struct TPMT_PUBLIC_PARMS_mst
768   {
769       UINT8     marshalType;
770       UINT8     elements;
771       UINT16    values[6];
772   } TPMT_PUBLIC_PARMS_mst;
773   typedef const struct TPMT_PUBLIC_mst
774   {
775       UINT8     marshalType;
776       UINT8     elements;
777       UINT16    values[18];
778   } TPMT_PUBLIC_mst;
779   typedef struct TPMU_SENSITIVE_COMPOSITE_mst
780   {
781       BYTE            countOfselectors;
782       BYTE            modifiers;
783       UINT16        offsetOfUnmarshalTypes;
784       UINT32        selectors[4];
785       UINT16        marshalingTypes[4];
786   } TPMU_SENSITIVE_COMPOSITE_mst;
787   typedef const struct TPMT_SENSITIVE_mst
788   {
789       UINT8     marshalType;
790       UINT8     elements;
791       UINT16    values[12];
792   } TPMT_SENSITIVE_mst;
793   typedef const struct _PRIVATE_mst
794   {
795       UINT8     marshalType;
796       UINT8     elements;
797       UINT16    values[9];
798   } _PRIVATE_mst;
799   typedef const struct TPMS_ID_OBJECT_mst
800   {
801       UINT8     marshalType;
802       UINT8     elements;
803       UINT16    values[6];
804   } TPMS_ID_OBJECT_mst;
805   typedef const struct TPMS_NV_PIN_COUNTER_PARAMETERS_mst
806   {
807       UINT8     marshalType;
808       UINT8     elements;
809       UINT16    values[6];
810   } TPMS_NV_PIN_COUNTER_PARAMETERS_mst;
811   typedef const struct TPMS_NV_PUBLIC_mst
812   {
813       UINT8     marshalType;
814       UINT8     elements;
815       UINT16    values[15];
816   } TPMS_NV_PUBLIC_mst;
817   typedef const struct TPMS_CONTEXT_DATA_mst
818   {
819       UINT8     marshalType;
```

```
820         UINT8     elements;
821         UINT16    values[6];
822     } TPMS_CONTEXT_DATA_mst;
823     typedef const struct TPMS_CONTEXT_mst
824     {
825         UINT8     marshalType;
826         UINT8     elements;
827         UINT16    values[12];
828     } TPMS_CONTEXT_mst;
829     typedef const struct TPMS_CREATION_DATA_mst
830     {
831         UINT8     marshalType;
832         UINT8     elements;
833         UINT16    values[21];
834     } TPMS_CREATION_DATA_mst;
835     typedef const struct TPM_AT_mst {
836         UINT8         marshalType;
837         UINT8         modifiers;
838         UINT8         errorCode;
839         UINT8         entries;
840         UINT32        values[4];
841     } TPM_AT_mst;
842     typedef const struct TPMS_AC_OUTPUT_mst
843     {
844         UINT8     marshalType;
845         UINT8     elements;
846         UINT16    values[6];
847     } TPMS_AC_OUTPUT_mst;
848     typedef const struct Type02_mst {
849         UINT8       marshalType;
850         UINT8       modifiers;
851         UINT8       errorCode;
852         UINT32      values[2];
853     } Type02_mst;
854     typedef const struct Type03_mst {
855         UINT8       marshalType;
856         UINT8       modifiers;
857         UINT8       errorCode;
858         UINT32      values[2];
859     } Type03_mst;
860     typedef const struct Type04_mst {
861         UINT8       marshalType;
862         UINT8       modifiers;
863         UINT8       errorCode;
864         UINT32      values[2];
865     } Type04_mst;
866     typedef const struct Type06_mst {
867         UINT8       marshalType;
868         UINT8       modifiers;
869         UINT8       errorCode;
870         UINT32      values[2];
871     } Type06_mst;
872     typedef const struct Type08_mst {
873         UINT8       marshalType;
874         UINT8       modifiers;
875         UINT8       errorCode;
876         UINT32      values[2];
877     } Type08_mst;
878     typedef const struct Type10_mst {
879         UINT8         marshalType;
880         UINT8         modifiers;
881         UINT8         errorCode;
882         UINT8         entries;
883         UINT32        values[1];
884     } Type10_mst;
885     typedef const struct Type11_mst {
```

```
886         UINT8           marshalType;
887         UINT8           modifiers;
888         UINT8           errorCode;
889         UINT8           entries;
890         UINT32          values[1];
891     } Type11_mst;
892     typedef const struct Type12_mst {
893         UINT8           marshalType;
894         UINT8           modifiers;
895         UINT8           errorCode;
896         UINT8           entries;
897         UINT32          values[2];
898     } Type12_mst;
899     typedef const struct Type13_mst {
900         UINT8           marshalType;
901         UINT8           modifiers;
902         UINT8           errorCode;
903         UINT8           entries;
904         UINT32          values[1];
905     } Type13_mst;
906     typedef const struct Type15_mst {
907         UINT8         marshalType;
908         UINT8         modifiers;
909         UINT8         errorCode;
910         UINT32        values[2];
911     } Type15_mst;
912     typedef const struct Type17_mst {
913         UINT8         marshalType;
914         UINT8         modifiers;
915         UINT8         errorCode;
916         UINT32        values[2];
917     } Type17_mst;
918     typedef const struct Type18_mst {
919         UINT8         marshalType;
920         UINT8         modifiers;
921         UINT8         errorCode;
922         UINT32        values[2];
923     } Type18_mst;
924     typedef const struct Type19_mst {
925         UINT8         marshalType;
926         UINT8         modifiers;
927         UINT8         errorCode;
928         UINT32        values[2];
929     } Type19_mst;
930     typedef const struct Type20_mst {
931         UINT8         marshalType;
932         UINT8         modifiers;
933         UINT8         errorCode;
934         UINT32        values[2];
935     } Type20_mst;
936     typedef const struct Type22_mst {
937         UINT8         marshalType;
938         UINT8         modifiers;
939         UINT8         errorCode;
940         UINT32        values[2];
941     } Type22_mst;
942     typedef const struct Type23_mst {
943         UINT8         marshalType;
944         UINT8         modifiers;
945         UINT8         errorCode;
946         UINT32        values[2];
947     } Type23_mst;
948     typedef const struct Type24_mst {
949         UINT8         marshalType;
950         UINT8         modifiers;
951         UINT8         errorCode;
```

```
952         UINT32      values[2];
953     } Type24_mst;
954     typedef const struct Type25_mst {
955         UINT8       marshalType;
956         UINT8       modifiers;
957         UINT8       errorCode;
958         UINT32      values[2];
959     } Type25_mst;
960     typedef const struct Type26_mst {
961         UINT8       marshalType;
962         UINT8       modifiers;
963         UINT8       errorCode;
964         UINT32      values[2];
965     } Type26_mst;
966     typedef const struct Type28_mst {
967         UINT8       marshalType;
968         UINT8       modifiers;
969         UINT8       errorCode;
970         UINT32      values[2];
971     } Type28_mst;
972     typedef const struct Type29_mst {
973         UINT8       marshalType;
974         UINT8       modifiers;
975         UINT8       errorCode;
976         UINT32      values[2];
977     } Type29_mst;
978     typedef const struct Type32_mst {
979         UINT8       marshalType;
980         UINT8       modifiers;
981         UINT8       errorCode;
982         UINT32      values[2];
983     } Type32_mst;
984     typedef const struct Type33_mst {
985         UINT8       marshalType;
986         UINT8       modifiers;
987         UINT8       errorCode;
988         UINT32      values[2];
989     } Type33_mst;
990     typedef const struct Type34_mst {
991         UINT8       marshalType;
992         UINT8       modifiers;
993         UINT8       errorCode;
994         UINT32      values[2];
995     } Type34_mst;
996     typedef const struct Type37_mst {
997         UINT8       marshalType;
998         UINT8       modifiers;
999         UINT8       errorCode;
1000        UINT32      values[2];
1001    } Type37_mst;
1002    typedef const struct Type40_mst {
1003        UINT8       marshalType;
1004        UINT8       modifiers;
1005        UINT8       errorCode;
1006        UINT32      values[2];
1007    } Type40_mst;
1008    typedef const struct Type41_mst {
1009        UINT8       marshalType;
1010        UINT8       modifiers;
1011        UINT8       errorCode;
1012        UINT32      values[2];
1013    } Type41_mst;
1014    typedef const struct Type43_mst {
1015        UINT8       marshalType;
1016        UINT8       modifiers;
1017        UINT8       errorCode;
```

```
1018        UINT32      values[2];
1019    } Type43_mst;
```

Defines for array lookup

```
1020    #define UINT8_ARRAY_MARSHAL_INDEX                   0   // 0x00
1021    #define TPM_CC_ARRAY_MARSHAL_INDEX                  1   // 0x01
1022    #define TPMA_CC_ARRAY_MARSHAL_INDEX                 2   // 0x02
1023    #define TPM_ALG_ID_ARRAY_MARSHAL_INDEX              3   // 0x03
1024    #define TPM_HANDLE_ARRAY_MARSHAL_INDEX              4   // 0x04
1025    #define TPM2B_DIGEST_ARRAY_MARSHAL_INDEX            5   // 0x05
1026    #define TPMT_HA_ARRAY_MARSHAL_INDEX                 6   // 0x06
1027    #define TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX      7   // 0x07
1028    #define TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX       8   // 0x08
1029    #define TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX    9   // 0x09
1030    #define TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX  10  // 0x0A
1031    #define TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX           11  // 0x0B
1032    #define TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX      12  // 0x0C
1033    #define TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX          13  // 0x0D
```

The defines to connect a typename to an index in the MarshalLookupTable()

```
1034    #define UINT8_MARSHAL_INDEX                         0   // 0x00
1035    #define BYTE_MARSHAL_INDEX                          UINT8_MARSHAL_INDEX
1036    #define TPM_HT_MARSHAL_INDEX                        UINT8_MARSHAL_INDEX
1037    #define TPMA_LOCALITY_MARSHAL_INDEX                 UINT8_MARSHAL_INDEX
1038    #define UINT16_MARSHAL_INDEX                        1   // 0x01
1039    #define TPM_KEY_SIZE_MARSHAL_INDEX                  UINT16_MARSHAL_INDEX
1040    #define TPM_KEY_BITS_MARSHAL_INDEX                  UINT16_MARSHAL_INDEX
1041    #define TPM_ALG_ID_MARSHAL_INDEX                    UINT16_MARSHAL_INDEX
1042    #define TPM_ST_MARSHAL_INDEX                        UINT16_MARSHAL_INDEX
1043    #define UINT32_MARSHAL_INDEX                        2   // 0x02
1044    #define TPM_ALGORITHM_ID_MARSHAL_INDEX              UINT32_MARSHAL_INDEX
1045    #define TPM_MODIFIER_INDICATOR_MARSHAL_INDEX        UINT32_MARSHAL_INDEX
1046    #define TPM_AUTHORIZATION_SIZE_MARSHAL_INDEX        UINT32_MARSHAL_INDEX
1047    #define TPM_PARAMETER_SIZE_MARSHAL_INDEX            UINT32_MARSHAL_INDEX
1048    #define TPM_SPEC_MARSHAL_INDEX                      UINT32_MARSHAL_INDEX
1049    #define TPM_GENERATED_MARSHAL_INDEX                 UINT32_MARSHAL_INDEX
1050    #define TPM_CC_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1051    #define TPM_RC_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1052    #define TPM_PT_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1053    #define TPM_PT_PCR_MARSHAL_INDEX                    UINT32_MARSHAL_INDEX
1054    #define TPM_PS_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1055    #define TPM_HANDLE_MARSHAL_INDEX                    UINT32_MARSHAL_INDEX
1056    #define TPM_RH_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1057    #define TPM_HC_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1058    #define TPMA_PERMANENT_MARSHAL_INDEX                UINT32_MARSHAL_INDEX
1059    #define TPMA_STARTUP_CLEAR_MARSHAL_INDEX            UINT32_MARSHAL_INDEX
1060    #define TPMA_MEMORY_MARSHAL_INDEX                   UINT32_MARSHAL_INDEX
1061    #define TPMA_CC_MARSHAL_INDEX                       UINT32_MARSHAL_INDEX
1062    #define TPMA_MODES_MARSHAL_INDEX                    UINT32_MARSHAL_INDEX
1063    #define TPMA_X509_KEY_USAGE_MARSHAL_INDEX           UINT32_MARSHAL_INDEX
1064    #define TPM_NV_INDEX_MARSHAL_INDEX                  UINT32_MARSHAL_INDEX
1065    #define TPM_AE_MARSHAL_INDEX                        UINT32_MARSHAL_INDEX
1066    #define UINT64_MARSHAL_INDEX                        3   // 0x03
1067    #define INT8_MARSHAL_INDEX                          4   // 0x04
1068    #define INT16_MARSHAL_INDEX                         5   // 0x05
1069    #define INT32_MARSHAL_INDEX                         6   // 0x06
1070    #define INT64_MARSHAL_INDEX                         7   // 0x07
1071    #define UINT0_MARSHAL_INDEX                         8   // 0x08
1072    #define TPM_ECC_CURVE_MARSHAL_INDEX                 9   // 0x09
1073    #define TPM_CLOCK_ADJUST_MARSHAL_INDEX              10  // 0x0A
1074    #define TPM_EO_MARSHAL_INDEX                        11  // 0x0B
1075    #define TPM_SU_MARSHAL_INDEX                        12  // 0x0C
1076    #define TPM_SE_MARSHAL_INDEX                        13  // 0x0D
```

```
1077  #define TPM_CAP_MARSHAL_INDEX                    14  // 0x0E
1078  #define TPMA_ALGORITHM_MARSHAL_INDEX             15  // 0x0F
1079  #define TPMA_OBJECT_MARSHAL_INDEX                16  // 0x10
1080  #define TPMA_SESSION_MARSHAL_INDEX               17  // 0x11
1081  #define TPMI_YES_NO_MARSHAL_INDEX                18  // 0x12
1082  #define TPMI_DH_OBJECT_MARSHAL_INDEX             19  // 0x13
1083  #define TPMI_DH_PARENT_MARSHAL_INDEX             20  // 0x14
1084  #define TPMI_DH_PERSISTENT_MARSHAL_INDEX         21  // 0x15
1085  #define TPMI_DH_ENTITY_MARSHAL_INDEX             22  // 0x16
1086  #define TPMI_DH_PCR_MARSHAL_INDEX                23  // 0x17
1087  #define TPMI_SH_AUTH_SESSION_MARSHAL_INDEX       24  // 0x18
1088  #define TPMI_SH_HMAC_MARSHAL_INDEX               25  // 0x19
1089  #define TPMI_SH_POLICY_MARSHAL_INDEX             26  // 0x1A
1090  #define TPMI_DH_CONTEXT_MARSHAL_INDEX            27  // 0x1B
1091  #define TPMI_DH_SAVED_MARSHAL_INDEX              28  // 0x1C
1092  #define TPMI_RH_HIERARCHY_MARSHAL_INDEX          29  // 0x1D
1093  #define TPMI_RH_ENABLES_MARSHAL_INDEX            30  // 0x1E
1094  #define TPMI_RH_HIERARCHY_AUTH_MARSHAL_INDEX     31  // 0x1F
1095  #define TPMI_RH_PLATFORM_MARSHAL_INDEX           32  // 0x20
1096  #define TPMI_RH_OWNER_MARSHAL_INDEX              33  // 0x21
1097  #define TPMI_RH_ENDORSEMENT_MARSHAL_INDEX        34  // 0x22
1098  #define TPMI_RH_PROVISION_MARSHAL_INDEX          35  // 0x23
1099  #define TPMI_RH_CLEAR_MARSHAL_INDEX              36  // 0x24
1100  #define TPMI_RH_NV_AUTH_MARSHAL_INDEX            37  // 0x25
1101  #define TPMI_RH_LOCKOUT_MARSHAL_INDEX            38  // 0x26
1102  #define TPMI_RH_NV_INDEX_MARSHAL_INDEX           39  // 0x27
1103  #define TPMI_RH_AC_MARSHAL_INDEX                 40  // 0x28
1104  #define TPMI_ALG_HASH_MARSHAL_INDEX              41  // 0x29
1105  #define TPMI_ALG_ASYM_MARSHAL_INDEX              42  // 0x2A
1106  #define TPMI_ALG_SYM_MARSHAL_INDEX               43  // 0x2B
1107  #define TPMI_ALG_SYM_OBJECT_MARSHAL_INDEX        44  // 0x2C
1108  #define TPMI_ALG_SYM_MODE_MARSHAL_INDEX          45  // 0x2D
1109  #define TPMI_ALG_KDF_MARSHAL_INDEX               46  // 0x2E
1110  #define TPMI_ALG_SIG_SCHEME_MARSHAL_INDEX        47  // 0x2F
1111  #define TPMI_ECC_KEY_EXCHANGE_MARSHAL_INDEX      48  // 0x30
1112  #define TPMI_ST_COMMAND_TAG_MARSHAL_INDEX        49  // 0x31
1113  #define TPMI_ALG_MAC_SCHEME_MARSHAL_INDEX        50  // 0x32
1114  #define TPMI_ALG_CIPHER_MODE_MARSHAL_INDEX       51  // 0x33
1115  #define TPMS_EMPTY_MARSHAL_INDEX                 52  // 0x34
1116  #define TPMS_ENC_SCHEME_RSAES_MARSHAL_INDEX      TPMS_EMPTY_MARSHAL_INDEX
1117  #define TPMS_ALGORITHM_DESCRIPTION_MARSHAL_INDEX 53  // 0x35
1118  #define TPMU_HA_MARSHAL_INDEX                    54  // 0x36
1119  #define TPMT_HA_MARSHAL_INDEX                    55  // 0x37
1120  #define TPM2B_DIGEST_MARSHAL_INDEX               56  // 0x38
1121  #define TPM2B_NONCE_MARSHAL_INDEX                TPM2B_DIGEST_MARSHAL_INDEX
1122  #define TPM2B_AUTH_MARSHAL_INDEX                 TPM2B_DIGEST_MARSHAL_INDEX
1123  #define TPM2B_OPERAND_MARSHAL_INDEX              TPM2B_DIGEST_MARSHAL_INDEX
1124  #define TPM2B_DATA_MARSHAL_INDEX                 57  // 0x39
1125  #define TPM2B_EVENT_MARSHAL_INDEX                58  // 0x3A
1126  #define TPM2B_MAX_BUFFER_MARSHAL_INDEX           59  // 0x3B
1127  #define TPM2B_MAX_NV_BUFFER_MARSHAL_INDEX        60  // 0x3C
1128  #define TPM2B_TIMEOUT_MARSHAL_INDEX              61  // 0x3D
1129  #define TPM2B_IV_MARSHAL_INDEX                   62  // 0x3E
1130  #define NULL_UNION_MARSHAL_INDEX                 63  // 0x3F
1131  #define TPMU_NAME_MARSHAL_INDEX                  NULL_UNION_MARSHAL_INDEX
1132  #define TPMU_SENSITIVE_CREATE_MARSHAL_INDEX      NULL_UNION_MARSHAL_INDEX
1133  #define TPM2B_NAME_MARSHAL_INDEX                 64  // 0x40
1134  #define TPMS_PCR_SELECT_MARSHAL_INDEX            65  // 0x41
1135  #define TPMS_PCR_SELECTION_MARSHAL_INDEX         66  // 0x42
1136  #define TPMT_TK_CREATION_MARSHAL_INDEX           67  // 0x43
1137  #define TPMT_TK_VERIFIED_MARSHAL_INDEX           68  // 0x44
1138  #define TPMT_TK_AUTH_MARSHAL_INDEX               69  // 0x45
1139  #define TPMT_TK_HASHCHECK_MARSHAL_INDEX          70  // 0x46
1140  #define TPMS_ALG_PROPERTY_MARSHAL_INDEX          71  // 0x47
1141  #define TPMS_TAGGED_PROPERTY_MARSHAL_INDEX       72  // 0x48
1142  #define TPMS_TAGGED_PCR_SELECT_MARSHAL_INDEX     73  // 0x49
```

```
#define TPMS_TAGGED_POLICY_MARSHAL_INDEX            74  // 0x4A
#define TPML_CC_MARSHAL_INDEX                       75  // 0x4B
#define TPML_CCA_MARSHAL_INDEX                      76  // 0x4C
#define TPML_ALG_MARSHAL_INDEX                      77  // 0x4D
#define TPML_HANDLE_MARSHAL_INDEX                   78  // 0x4E
#define TPML_DIGEST_MARSHAL_INDEX                   79  // 0x4F
#define TPML_DIGEST_VALUES_MARSHAL_INDEX            80  // 0x50
#define TPML_PCR_SELECTION_MARSHAL_INDEX            81  // 0x51
#define TPML_ALG_PROPERTY_MARSHAL_INDEX             82  // 0x52
#define TPML_TAGGED_TPM_PROPERTY_MARSHAL_INDEX      83  // 0x53
#define TPML_TAGGED_PCR_PROPERTY_MARSHAL_INDEX      84  // 0x54
#define TPML_ECC_CURVE_MARSHAL_INDEX                85  // 0x55
#define TPML_TAGGED_POLICY_MARSHAL_INDEX            86  // 0x56
#define TPMU_CAPABILITIES_MARSHAL_INDEX             87  // 0x57
#define TPMS_CAPABILITY_DATA_MARSHAL_INDEX          88  // 0x58
#define TPMS_CLOCK_INFO_MARSHAL_INDEX               89  // 0x59
#define TPMS_TIME_INFO_MARSHAL_INDEX                90  // 0x5A
#define TPMS_TIME_ATTEST_INFO_MARSHAL_INDEX         91  // 0x5B
#define TPMS_CERTIFY_INFO_MARSHAL_INDEX             92  // 0x5C
#define TPMS_QUOTE_INFO_MARSHAL_INDEX               93  // 0x5D
#define TPMS_COMMAND_AUDIT_INFO_MARSHAL_INDEX       94  // 0x5E
#define TPMS_SESSION_AUDIT_INFO_MARSHAL_INDEX       95  // 0x5F
#define TPMS_CREATION_INFO_MARSHAL_INDEX            96  // 0x60
#define TPMS_NV_CERTIFY_INFO_MARSHAL_INDEX          97  // 0x61
#define TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_INDEX   98  // 0x62
#define TPMI_ST_ATTEST_MARSHAL_INDEX                99  // 0x63
#define TPMU_ATTEST_MARSHAL_INDEX                   100 // 0x64
#define TPMS_ATTEST_MARSHAL_INDEX                   101 // 0x65
#define TPM2B_ATTEST_MARSHAL_INDEX                  102 // 0x66
#define TPMS_AUTH_COMMAND_MARSHAL_INDEX             103 // 0x67
#define TPMS_AUTH_RESPONSE_MARSHAL_INDEX            104 // 0x68
#define TPMI_TDES_KEY_BITS_MARSHAL_INDEX            105 // 0x69
#define TPMI_AES_KEY_BITS_MARSHAL_INDEX             106 // 0x6A
#define TPMI_SM4_KEY_BITS_MARSHAL_INDEX             107 // 0x6B
#define TPMI_CAMELLIA_KEY_BITS_MARSHAL_INDEX        108 // 0x6C
#define TPMU_SYM_KEY_BITS_MARSHAL_INDEX             109 // 0x6D
#define TPMU_SYM_MODE_MARSHAL_INDEX                 110 // 0x6E
#define TPMT_SYM_DEF_MARSHAL_INDEX                  111 // 0x6F
#define TPMT_SYM_DEF_OBJECT_MARSHAL_INDEX           112 // 0x70
#define TPM2B_SYM_KEY_MARSHAL_INDEX                 113 // 0x71
#define TPMS_SYMCIPHER_PARMS_MARSHAL_INDEX          114 // 0x72
#define TPM2B_LABEL_MARSHAL_INDEX                   115 // 0x73
#define TPMS_DERIVE_MARSHAL_INDEX                   116 // 0x74
#define TPM2B_DERIVE_MARSHAL_INDEX                  117 // 0x75
#define TPM2B_SENSITIVE_DATA_MARSHAL_INDEX          118 // 0x76
#define TPMS_SENSITIVE_CREATE_MARSHAL_INDEX         119 // 0x77
#define TPM2B_SENSITIVE_CREATE_MARSHAL_INDEX        120 // 0x78
#define TPMS_SCHEME_HASH_MARSHAL_INDEX              121 // 0x79
#define TPMS_SCHEME_HMAC_MARSHAL_INDEX              TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SIG_SCHEME_RSASSA_MARSHAL_INDEX        TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SIG_SCHEME_RSAPSS_MARSHAL_INDEX        TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SIG_SCHEME_ECDSA_MARSHAL_INDEX         TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SIG_SCHEME_SM2_MARSHAL_INDEX           TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_INDEX     TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_ENC_SCHEME_OAEP_MARSHAL_INDEX          TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_KEY_SCHEME_ECDH_MARSHAL_INDEX          TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_KEY_SCHEME_ECMQV_MARSHAL_INDEX         TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SCHEME_MGF1_MARSHAL_INDEX              TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_INDEX    TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SCHEME_KDF2_MARSHAL_INDEX              TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SCHEME_KDF1_SP800_108_MARSHAL_INDEX    TPMS_SCHEME_HASH_MARSHAL_INDEX
#define TPMS_SCHEME_ECDAA_MARSHAL_INDEX             122 // 0x7A
#define TPMS_SIG_SCHEME_ECDAA_MARSHAL_INDEX         TPMS_SCHEME_ECDAA_MARSHAL_INDEX
#define TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_INDEX     123 // 0x7B
#define TPMS_SCHEME_XOR_MARSHAL_INDEX               124 // 0x7C
#define TPMU_SCHEME_KEYEDHASH_MARSHAL_INDEX         125 // 0x7D
```

```
1209  #define TPMT_KEYEDHASH_SCHEME_MARSHAL_INDEX          126 // 0x7E
1210  #define TPMU_SIG_SCHEME_MARSHAL_INDEX                127 // 0x7F
1211  #define TPMT_SIG_SCHEME_MARSHAL_INDEX                128 // 0x80
1212  #define TPMU_KDF_SCHEME_MARSHAL_INDEX                129 // 0x81
1213  #define TPMT_KDF_SCHEME_MARSHAL_INDEX                130 // 0x82
1214  #define TPMI_ALG_ASYM_SCHEME_MARSHAL_INDEX           131 // 0x83
1215  #define TPMU_ASYM_SCHEME_MARSHAL_INDEX               132 // 0x84
1216  #define TPMT_ASYM_SCHEME_MARSHAL_INDEX               133 // 0x85
1217  #define TPMI_ALG_RSA_SCHEME_MARSHAL_INDEX            134 // 0x86
1218  #define TPMT_RSA_SCHEME_MARSHAL_INDEX                135 // 0x87
1219  #define TPMI_ALG_RSA_DECRYPT_MARSHAL_INDEX           136 // 0x88
1220  #define TPMT_RSA_DECRYPT_MARSHAL_INDEX               137 // 0x89
1221  #define TPM2B_PUBLIC_KEY_RSA_MARSHAL_INDEX           138 // 0x8A
1222  #define TPMI_RSA_KEY_BITS_MARSHAL_INDEX              139 // 0x8B
1223  #define TPM2B_PRIVATE_KEY_RSA_MARSHAL_INDEX          140 // 0x8C
1224  #define TPM2B_ECC_PARAMETER_MARSHAL_INDEX            141 // 0x8D
1225  #define TPMS_ECC_POINT_MARSHAL_INDEX                 142 // 0x8E
1226  #define TPM2B_ECC_POINT_MARSHAL_INDEX                143 // 0x8F
1227  #define TPMI_ALG_ECC_SCHEME_MARSHAL_INDEX            144 // 0x90
1228  #define TPMI_ECC_CURVE_MARSHAL_INDEX                 145 // 0x91
1229  #define TPMT_ECC_SCHEME_MARSHAL_INDEX                146 // 0x92
1230  #define TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_INDEX      147 // 0x93
1231  #define TPMS_SIGNATURE_RSA_MARSHAL_INDEX             148 // 0x94
1232  #define TPMS_SIGNATURE_RSASSA_MARSHAL_INDEX          TPMS_SIGNATURE_RSA_MARSHAL_INDEX
1233  #define TPMS_SIGNATURE_RSAPSS_MARSHAL_INDEX          TPMS_SIGNATURE_RSA_MARSHAL_INDEX
1234  #define TPMS_SIGNATURE_ECC_MARSHAL_INDEX             149 // 0x95
1235  #define TPMS_SIGNATURE_ECDAA_MARSHAL_INDEX           TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1236  #define TPMS_SIGNATURE_ECDSA_MARSHAL_INDEX           TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1237  #define TPMS_SIGNATURE_SM2_MARSHAL_INDEX             TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1238  #define TPMS_SIGNATURE_ECSCHNORR_MARSHAL_INDEX       TPMS_SIGNATURE_ECC_MARSHAL_INDEX
1239  #define TPMU_SIGNATURE_MARSHAL_INDEX                 150 // 0x96
1240  #define TPMT_SIGNATURE_MARSHAL_INDEX                 151 // 0x97
1241  #define TPMU_ENCRYPTED_SECRET_MARSHAL_INDEX          152 // 0x98
1242  #define TPM2B_ENCRYPTED_SECRET_MARSHAL_INDEX         153 // 0x99
1243  #define TPMI_ALG_PUBLIC_MARSHAL_INDEX                154 // 0x9A
1244  #define TPMU_PUBLIC_ID_MARSHAL_INDEX                 155 // 0x9B
1245  #define TPMS_KEYEDHASH_PARMS_MARSHAL_INDEX           156 // 0x9C
1246  #define TPMS_ASYM_PARMS_MARSHAL_INDEX                157 // 0x9D
1247  #define TPMS_RSA_PARMS_MARSHAL_INDEX                 158 // 0x9E
1248  #define TPMS_ECC_PARMS_MARSHAL_INDEX                 159 // 0x9F
1249  #define TPMU_PUBLIC_PARMS_MARSHAL_INDEX              160 // 0xA0
1250  #define TPMT_PUBLIC_PARMS_MARSHAL_INDEX              161 // 0xA1
1251  #define TPMT_PUBLIC_MARSHAL_INDEX                    162 // 0xA2
1252  #define TPM2B_PUBLIC_MARSHAL_INDEX                   163 // 0xA3
1253  #define TPM2B_TEMPLATE_MARSHAL_INDEX                 164 // 0xA4
1254  #define TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_INDEX  165 // 0xA5
1255  #define TPMU_SENSITIVE_COMPOSITE_MARSHAL_INDEX       166 // 0xA6
1256  #define TPMT_SENSITIVE_MARSHAL_INDEX                 167 // 0xA7
1257  #define TPM2B_SENSITIVE_MARSHAL_INDEX                168 // 0xA8
1258  #define _PRIVATE_MARSHAL_INDEX                       169 // 0xA9
1259  #define TPM2B_PRIVATE_MARSHAL_INDEX                  170 // 0xAA
1260  #define TPMS_ID_OBJECT_MARSHAL_INDEX                 171 // 0xAB
1261  #define TPM2B_ID_OBJECT_MARSHAL_INDEX                172 // 0xAC
1262  #define TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_INDEX  173 // 0xAD
1263  #define TPMA_NV_MARSHAL_INDEX                        174 // 0xAE
1264  #define TPMS_NV_PUBLIC_MARSHAL_INDEX                 175 // 0xAF
1265  #define TPM2B_NV_PUBLIC_MARSHAL_INDEX                176 // 0xB0
1266  #define TPM2B_CONTEXT_SENSITIVE_MARSHAL_INDEX        177 // 0xB1
1267  #define TPMS_CONTEXT_DATA_MARSHAL_INDEX              178 // 0xB2
1268  #define TPM2B_CONTEXT_DATA_MARSHAL_INDEX             179 // 0xB3
1269  #define TPMS_CONTEXT_MARSHAL_INDEX                   180 // 0xB4
1270  #define TPMS_CREATION_DATA_MARSHAL_INDEX             181 // 0xB5
1271  #define TPM2B_CREATION_DATA_MARSHAL_INDEX            182 // 0xB6
1272  #define TPM_AT_MARSHAL_INDEX                         183 // 0xB7
1273  #define TPMS_AC_OUTPUT_MARSHAL_INDEX                 184 // 0xB8
1274  #define TPML_AC_CAPABILITIES_MARSHAL_INDEX           185 // 0xB9
```

```
1275    #define Type00_MARSHAL_INDEX                    186 // 0xBA
1276    #define Type01_MARSHAL_INDEX                    187 // 0xBB
1277    #define Type02_MARSHAL_INDEX                    188 // 0xBC
1278    #define Type03_MARSHAL_INDEX                    189 // 0xBD
1279    #define Type04_MARSHAL_INDEX                    190 // 0xBE
1280    #define Type05_MARSHAL_INDEX                    191 // 0xBF
1281    #define Type06_MARSHAL_INDEX                    192 // 0xC0
1282    #define Type07_MARSHAL_INDEX                    193 // 0xC1
1283    #define Type08_MARSHAL_INDEX                    194 // 0xC2
1284    #define Type09_MARSHAL_INDEX                    Type08_MARSHAL_INDEX
1285    #define Type14_MARSHAL_INDEX                    Type08_MARSHAL_INDEX
1286    #define Type10_MARSHAL_INDEX                    195 // 0xC3
1287    #define Type11_MARSHAL_INDEX                    196 // 0xC4
1288    #define Type12_MARSHAL_INDEX                    197 // 0xC5
1289    #define Type13_MARSHAL_INDEX                    198 // 0xC6
1290    #define Type15_MARSHAL_INDEX                    199 // 0xC7
1291    #define Type16_MARSHAL_INDEX                    Type15_MARSHAL_INDEX
1292    #define Type17_MARSHAL_INDEX                    200 // 0xC8
1293    #define Type18_MARSHAL_INDEX                    201 // 0xC9
1294    #define Type19_MARSHAL_INDEX                    202 // 0xCA
1295    #define Type20_MARSHAL_INDEX                    203 // 0xCB
1296    #define Type21_MARSHAL_INDEX                    Type20_MARSHAL_INDEX
1297    #define Type22_MARSHAL_INDEX                    204 // 0xCC
1298    #define Type23_MARSHAL_INDEX                    205 // 0xCD
1299    #define Type24_MARSHAL_INDEX                    206 // 0xCE
1300    #define Type25_MARSHAL_INDEX                    207 // 0xCF
1301    #define Type26_MARSHAL_INDEX                    208 // 0xD0
1302    #define Type27_MARSHAL_INDEX                    209 // 0xD1
1303    #define Type28_MARSHAL_INDEX                    210 // 0xD2
1304    #define Type29_MARSHAL_INDEX                    211 // 0xD3
1305    #define Type30_MARSHAL_INDEX                    212 // 0xD4
1306    #define Type31_MARSHAL_INDEX                    213 // 0xD5
1307    #define Type32_MARSHAL_INDEX                    214 // 0xD6
1308    #define Type33_MARSHAL_INDEX                    215 // 0xD7
1309    #define Type34_MARSHAL_INDEX                    216 // 0xD8
1310    #define Type35_MARSHAL_INDEX                    217 // 0xD9
1311    #define Type36_MARSHAL_INDEX                    218 // 0xDA
1312    #define Type37_MARSHAL_INDEX                    219 // 0xDB
1313    #define Type38_MARSHAL_INDEX                    220 // 0xDC
1314    #define Type39_MARSHAL_INDEX                    221 // 0xDD
1315    #define Type40_MARSHAL_INDEX                    222 // 0xDE
1316    #define Type41_MARSHAL_INDEX                    223 // 0xDF
1317    #define Type42_MARSHAL_INDEX                    224 // 0xE0
1318    #define Type43_MARSHAL_INDEX                    225 // 0xE1
1319    //#defines to change calling sequence for code using marshaling
1320    #define UINT8_Unmarshal(target, buffer, size)                                   \
1321        Unmarshal(UINT8_MARSHAL_INDEX, (target), (buffer), (size))
1322    #define UINT8_Marshal(source, buffer, size)                                     \
1323        Marshal(UINT8_MARSHAL_INDEX, (source), (buffer), (size))
1324    #define BYTE_Unmarshal(target, buffer, size)                                    \
1325        Unmarshal(UINT8_MARSHAL_INDEX, (target), (buffer), (size))
1326    #define BYTE_Marshal(source, buffer, size)                                      \
1327        Marshal(UINT8_MARSHAL_INDEX, (source), (buffer), (size))
1328    #define INT8_Unmarshal(target, buffer, size)                                    \
1329        Unmarshal(INT8_MARSHAL_INDEX, (target), (buffer), (size))
1330    #define INT8_Marshal(source, buffer, size)                                      \
1331        Marshal(INT8_MARSHAL_INDEX, (source), (buffer), (size))
1332    #define UINT16_Unmarshal(target, buffer, size)                                  \
1333        Unmarshal(UINT16_MARSHAL_INDEX, (target), (buffer), (size))
1334    #define UINT16_Marshal(source, buffer, size)                                    \
1335        Marshal(UINT16_MARSHAL_INDEX, (source), (buffer), (size))
1336    #define INT16_Unmarshal(target, buffer, size)                                   \
1337        Unmarshal(INT16_MARSHAL_INDEX, (target), (buffer), (size))
1338    #define INT16_Marshal(source, buffer, size)                                     \
1339        Marshal(INT16_MARSHAL_INDEX, (source), (buffer), (size))
1340    #define UINT32_Unmarshal(target, buffer, size)                                  \
```

```
1341        Unmarshal(UINT32_MARSHAL_INDEX, (target), (buffer), (size))
1342    #define UINT32_Marshal(source, buffer, size)                                \
1343        Marshal(UINT32_MARSHAL_INDEX, (source), (buffer), (size))
1344    #define INT32_Unmarshal(target, buffer, size)                               \
1345        Unmarshal(INT32_MARSHAL_INDEX, (target), (buffer), (size))
1346    #define INT32_Marshal(source, buffer, size)                                 \
1347        Marshal(INT32_MARSHAL_INDEX, (source), (buffer), (size))
1348    #define UINT64_Unmarshal(target, buffer, size)                              \
1349        Unmarshal(UINT64_MARSHAL_INDEX, (target), (buffer), (size))
1350    #define UINT64_Marshal(source, buffer, size)                                \
1351        Marshal(UINT64_MARSHAL_INDEX, (source), (buffer), (size))
1352    #define INT64_Unmarshal(target, buffer, size)                               \
1353        Unmarshal(INT64_MARSHAL_INDEX, (target), (buffer), (size))
1354    #define INT64_Marshal(source, buffer, size)                                 \
1355        Marshal(INT64_MARSHAL_INDEX, (source), (buffer), (size))
1356    #define TPM_ALGORITHM_ID_Unmarshal(target, buffer, size)                    \
1357        Unmarshal(TPM_ALGORITHM_ID_MARSHAL_INDEX, (target), (buffer), (size))
1358    #define TPM_ALGORITHM_ID_Marshal(source, buffer, size)                      \
1359        Marshal(TPM_ALGORITHM_ID_MARSHAL_INDEX, (source), (buffer), (size))
1360    #define TPM_MODIFIER_INDICATOR_Unmarshal(target, buffer, size)              \
1361        Unmarshal(TPM_MODIFIER_INDICATOR_MARSHAL_INDEX, (target), (buffer), (size))
1362    #define TPM_MODIFIER_INDICATOR_Marshal(source, buffer, size)                \
1363        Marshal(TPM_MODIFIER_INDICATOR_MARSHAL_INDEX, (source), (buffer), (size))
1364    #define TPM_AUTHORIZATION_SIZE_Unmarshal(target, buffer, size)              \
1365        Unmarshal(TPM_AUTHORIZATION_SIZE_MARSHAL_INDEX, (target), (buffer), (size))
1366    #define TPM_AUTHORIZATION_SIZE_Marshal(source, buffer, size)                \
1367        Marshal(TPM_AUTHORIZATION_SIZE_MARSHAL_INDEX, (source), (buffer), (size))
1368    #define TPM_PARAMETER_SIZE_Unmarshal(target, buffer, size)                  \
1369        Unmarshal(TPM_PARAMETER_SIZE_MARSHAL_INDEX, (target), (buffer), (size))
1370    #define TPM_PARAMETER_SIZE_Marshal(source, buffer, size)                    \
1371        Marshal(TPM_PARAMETER_SIZE_MARSHAL_INDEX, (source), (buffer), (size))
1372    #define TPM_KEY_SIZE_Unmarshal(target, buffer, size)                        \
1373        Unmarshal(TPM_KEY_SIZE_MARSHAL_INDEX, (target), (buffer), (size))
1374    #define TPM_KEY_SIZE_Marshal(source, buffer, size)                          \
1375        Marshal(TPM_KEY_SIZE_MARSHAL_INDEX, (source), (buffer), (size))
1376    #define TPM_KEY_BITS_Unmarshal(target, buffer, size)                        \
1377        Unmarshal(TPM_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1378    #define TPM_KEY_BITS_Marshal(source, buffer, size)                          \
1379        Marshal(TPM_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1380    #define TPM_GENERATED_Marshal(source, buffer, size)                         \
1381        Marshal(TPM_GENERATED_MARSHAL_INDEX, (source), (buffer), (size))
1382    #define TPM_ALG_ID_Unmarshal(target, buffer, size)                          \
1383        Unmarshal(TPM_ALG_ID_MARSHAL_INDEX, (target), (buffer), (size))
1384    #define TPM_ALG_ID_Marshal(source, buffer, size)                            \
1385        Marshal(TPM_ALG_ID_MARSHAL_INDEX, (source), (buffer), (size))
1386    #define TPM_ECC_CURVE_Unmarshal(target, buffer, size)                       \
1387        Unmarshal(TPM_ECC_CURVE_MARSHAL_INDEX, (target), (buffer), (size))
1388    #define TPM_ECC_CURVE_Marshal(source, buffer, size)                         \
1389        Marshal(TPM_ECC_CURVE_MARSHAL_INDEX, (source), (buffer), (size))
1390    #define TPM_CC_Unmarshal(target, buffer, size)                              \
1391        Unmarshal(TPM_CC_MARSHAL_INDEX, (target), (buffer), (size))
1392    #define TPM_CC_Marshal(source, buffer, size)                                \
1393        Marshal(TPM_CC_MARSHAL_INDEX, (source), (buffer), (size))
1394    #define TPM_RC_Marshal(source, buffer, size)                                \
1395        Marshal(TPM_RC_MARSHAL_INDEX, (source), (buffer), (size))
1396    #define TPM_CLOCK_ADJUST_Unmarshal(target, buffer, size)                    \
1397        Unmarshal(TPM_CLOCK_ADJUST_MARSHAL_INDEX, (target), (buffer), (size))
1398    #define TPM_EO_Unmarshal(target, buffer, size)                              \
1399        Unmarshal(TPM_EO_MARSHAL_INDEX, (target), (buffer), (size))
1400    #define TPM_EO_Marshal(source, buffer, size)                                \
1401        Marshal(TPM_EO_MARSHAL_INDEX, (source), (buffer), (size))
1402    #define TPM_ST_Unmarshal(target, buffer, size)                              \
1403        Unmarshal(TPM_ST_MARSHAL_INDEX, (target), (buffer), (size))
1404    #define TPM_ST_Marshal(source, buffer, size)                                \
1405        Marshal(TPM_ST_MARSHAL_INDEX, (source), (buffer), (size))
1406    #define TPM_SU_Unmarshal(target, buffer, size)                              \
```

```
1407        Unmarshal(TPM_SU_MARSHAL_INDEX, (target), (buffer), (size))
1408    #define TPM_SE_Unmarshal(target, buffer, size)                          \
1409        Unmarshal(TPM_SE_MARSHAL_INDEX, (target), (buffer), (size))
1410    #define TPM_CAP_Unmarshal(target, buffer, size)                         \
1411        Unmarshal(TPM_CAP_MARSHAL_INDEX, (target), (buffer), (size))
1412    #define TPM_CAP_Marshal(source, buffer, size)                           \
1413        Marshal(TPM_CAP_MARSHAL_INDEX, (source), (buffer), (size))
1414    #define TPM_PT_Unmarshal(target, buffer, size)                          \
1415        Unmarshal(TPM_PT_MARSHAL_INDEX, (target), (buffer), (size))
1416    #define TPM_PT_Marshal(source, buffer, size)                            \
1417        Marshal(TPM_PT_MARSHAL_INDEX, (source), (buffer), (size))
1418    #define TPM_PT_PCR_Unmarshal(target, buffer, size)                      \
1419        Unmarshal(TPM_PT_PCR_MARSHAL_INDEX, (target), (buffer), (size))
1420    #define TPM_PT_PCR_Marshal(source, buffer, size)                        \
1421        Marshal(TPM_PT_PCR_MARSHAL_INDEX, (source), (buffer), (size))
1422    #define TPM_PS_Marshal(source, buffer, size)                            \
1423        Marshal(TPM_PS_MARSHAL_INDEX, (source), (buffer), (size))
1424    #define TPM_HANDLE_Unmarshal(target, buffer, size)                      \
1425        Unmarshal(TPM_HANDLE_MARSHAL_INDEX, (target), (buffer), (size))
1426    #define TPM_HANDLE_Marshal(source, buffer, size)                        \
1427        Marshal(TPM_HANDLE_MARSHAL_INDEX, (source), (buffer), (size))
1428    #define TPM_HT_Unmarshal(target, buffer, size)                          \
1429        Unmarshal(TPM_HT_MARSHAL_INDEX, (target), (buffer), (size))
1430    #define TPM_HT_Marshal(source, buffer, size)                            \
1431        Marshal(TPM_HT_MARSHAL_INDEX, (source), (buffer), (size))
1432    #define TPM_RH_Unmarshal(target, buffer, size)                          \
1433        Unmarshal(TPM_RH_MARSHAL_INDEX, (target), (buffer), (size))
1434    #define TPM_RH_Marshal(source, buffer, size)                            \
1435        Marshal(TPM_RH_MARSHAL_INDEX, (source), (buffer), (size))
1436    #define TPM_HC_Unmarshal(target, buffer, size)                          \
1437        Unmarshal(TPM_HC_MARSHAL_INDEX, (target), (buffer), (size))
1438    #define TPM_HC_Marshal(source, buffer, size)                            \
1439        Marshal(TPM_HC_MARSHAL_INDEX, (source), (buffer), (size))
1440    #define TPMA_ALGORITHM_Unmarshal(target, buffer, size)                  \
1441        Unmarshal(TPMA_ALGORITHM_MARSHAL_INDEX, (target), (buffer), (size))
1442    #define TPMA_ALGORITHM_Marshal(source, buffer, size)                    \
1443        Marshal(TPMA_ALGORITHM_MARSHAL_INDEX, (source), (buffer), (size))
1444    #define TPMA_OBJECT_Unmarshal(target, buffer, size)                     \
1445        Unmarshal(TPMA_OBJECT_MARSHAL_INDEX, (target), (buffer), (size))
1446    #define TPMA_OBJECT_Marshal(source, buffer, size)                       \
1447        Marshal(TPMA_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1448    #define TPMA_SESSION_Unmarshal(target, buffer, size)                    \
1449        Unmarshal(TPMA_SESSION_MARSHAL_INDEX, (target), (buffer), (size))
1450    #define TPMA_SESSION_Marshal(source, buffer, size)                      \
1451        Marshal(TPMA_SESSION_MARSHAL_INDEX, (source), (buffer), (size))
1452    #define TPMA_LOCALITY_Unmarshal(target, buffer, size)                   \
1453        Unmarshal(TPMA_LOCALITY_MARSHAL_INDEX, (target), (buffer), (size))
1454    #define TPMA_LOCALITY_Marshal(source, buffer, size)                     \
1455        Marshal(TPMA_LOCALITY_MARSHAL_INDEX, (source), (buffer), (size))
1456    #define TPMA_PERMANENT_Marshal(source, buffer, size)                    \
1457        Marshal(TPMA_PERMANENT_MARSHAL_INDEX, (source), (buffer), (size))
1458    #define TPMA_STARTUP_CLEAR_Marshal(source, buffer, size)                \
1459        Marshal(TPMA_STARTUP_CLEAR_MARSHAL_INDEX, (source), (buffer), (size))
1460    #define TPMA_MEMORY_Marshal(source, buffer, size)                       \
1461        Marshal(TPMA_MEMORY_MARSHAL_INDEX, (source), (buffer), (size))
1462    #define TPMA_CC_Marshal(source, buffer, size)                           \
1463        Marshal(TPMA_CC_MARSHAL_INDEX, (source), (buffer), (size))
1464    #define TPMA_MODES_Marshal(source, buffer, size)                        \
1465        Marshal(TPMA_MODES_MARSHAL_INDEX, (source), (buffer), (size))
1466    #define TPMA_X509_KEY_USAGE_Marshal(source, buffer, size)               \
1467        Marshal(TPMA_X509_KEY_USAGE_MARSHAL_INDEX, (source), (buffer), (size))
1468    #define TPMI_YES_NO_Unmarshal(target, buffer, size)                     \
1469        Unmarshal(TPMI_YES_NO_MARSHAL_INDEX, (target), (buffer), (size))
1470    #define TPMI_YES_NO_Marshal(source, buffer, size)                       \
1471        Marshal(TPMI_YES_NO_MARSHAL_INDEX, (source), (buffer), (size))
1472    #define TPMI_DH_OBJECT_Unmarshal(target, buffer, size, flag)            \
```

```
1473        Unmarshal(TPMI_DH_OBJECT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1474          (buffer), (size))
1475    #define TPMI_DH_OBJECT_Marshal(source, buffer, size)                                  \
1476        Marshal(TPMI_DH_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1477    #define TPMI_DH_PARENT_Unmarshal(target, buffer, size, flag)                          \
1478        Unmarshal(TPMI_DH_PARENT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1479          (buffer), (size))
1480    #define TPMI_DH_PARENT_Marshal(source, buffer, size)                                  \
1481        Marshal(TPMI_DH_PARENT_MARSHAL_INDEX, (source), (buffer), (size))
1482    #define TPMI_DH_PERSISTENT_Unmarshal(target, buffer, size)                            \
1483        Unmarshal(TPMI_DH_PERSISTENT_MARSHAL_INDEX, (target), (buffer), (size))
1484    #define TPMI_DH_PERSISTENT_Marshal(source, buffer, size)                              \
1485        Marshal(TPMI_DH_PERSISTENT_MARSHAL_INDEX, (source), (buffer), (size))
1486    #define TPMI_DH_ENTITY_Unmarshal(target, buffer, size, flag)                          \
1487        Unmarshal(TPMI_DH_ENTITY_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),      \
1488          (buffer), (size))
1489    #define TPMI_DH_PCR_Unmarshal(target, buffer, size, flag)                             \
1490        Unmarshal(TPMI_DH_PCR_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),         \
1491          (buffer), (size))
1492    #define TPMI_SH_AUTH_SESSION_Unmarshal(target, buffer, size, flag)                    \
1493        Unmarshal(TPMI_SH_AUTH_SESSION_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1494          (buffer), (size))
1495    #define TPMI_SH_AUTH_SESSION_Marshal(source, buffer, size)                            \
1496        Marshal(TPMI_SH_AUTH_SESSION_MARSHAL_INDEX, (source), (buffer), (size))
1497    #define TPMI_SH_HMAC_Unmarshal(target, buffer, size)                                  \
1498        Unmarshal(TPMI_SH_HMAC_MARSHAL_INDEX, (target), (buffer), (size))
1499    #define TPMI_SH_HMAC_Marshal(source, buffer, size)                                    \
1500        Marshal(TPMI_SH_HMAC_MARSHAL_INDEX, (source), (buffer), (size))
1501    #define TPMI_SH_POLICY_Unmarshal(target, buffer, size)                                \
1502        Unmarshal(TPMI_SH_POLICY_MARSHAL_INDEX, (target), (buffer), (size))
1503    #define TPMI_SH_POLICY_Marshal(source, buffer, size)                                  \
1504        Marshal(TPMI_SH_POLICY_MARSHAL_INDEX, (source), (buffer), (size))
1505    #define TPMI_DH_CONTEXT_Unmarshal(target, buffer, size)                               \
1506        Unmarshal(TPMI_DH_CONTEXT_MARSHAL_INDEX, (target), (buffer), (size))
1507    #define TPMI_DH_CONTEXT_Marshal(source, buffer, size)                                 \
1508        Marshal(TPMI_DH_CONTEXT_MARSHAL_INDEX, (source), (buffer), (size))
1509    #define TPMI_DH_SAVED_Unmarshal(target, buffer, size)                                 \
1510        Unmarshal(TPMI_DH_SAVED_MARSHAL_INDEX, (target), (buffer), (size))
1511    #define TPMI_DH_SAVED_Marshal(source, buffer, size)                                   \
1512        Marshal(TPMI_DH_SAVED_MARSHAL_INDEX, (source), (buffer), (size))
1513    #define TPMI_RH_HIERARCHY_Unmarshal(target, buffer, size, flag)                       \
1514        Unmarshal(TPMI_RH_HIERARCHY_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1515          (buffer), (size))
1516    #define TPMI_RH_HIERARCHY_Marshal(source, buffer, size)                               \
1517        Marshal(TPMI_RH_HIERARCHY_MARSHAL_INDEX, (source), (buffer), (size))
1518    #define TPMI_RH_ENABLES_Unmarshal(target, buffer, size, flag)                         \
1519        Unmarshal(TPMI_RH_ENABLES_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),     \
1520          (buffer), (size))
1521    #define TPMI_RH_ENABLES_Marshal(source, buffer, size)                                 \
1522        Marshal(TPMI_RH_ENABLES_MARSHAL_INDEX, (source), (buffer), (size))
1523    #define TPMI_RH_HIERARCHY_AUTH_Unmarshal(target, buffer, size)                        \
1524        Unmarshal(TPMI_RH_HIERARCHY_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
1525    #define TPMI_RH_PLATFORM_Unmarshal(target, buffer, size)                              \
1526        Unmarshal(TPMI_RH_PLATFORM_MARSHAL_INDEX, (target), (buffer), (size))
1527    #define TPMI_RH_OWNER_Unmarshal(target, buffer, size, flag)                           \
1528        Unmarshal(TPMI_RH_OWNER_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),       \
1529          (buffer), (size))
1530    #define TPMI_RH_ENDORSEMENT_Unmarshal(target, buffer, size, flag)                     \
1531        Unmarshal(TPMI_RH_ENDORSEMENT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1532          (buffer), (size))
1533    #define TPMI_RH_PROVISION_Unmarshal(target, buffer, size)                             \
1534        Unmarshal(TPMI_RH_PROVISION_MARSHAL_INDEX, (target), (buffer), (size))
1535    #define TPMI_RH_CLEAR_Unmarshal(target, buffer, size)                                 \
1536        Unmarshal(TPMI_RH_CLEAR_MARSHAL_INDEX, (target), (buffer), (size))
1537    #define TPMI_RH_NV_AUTH_Unmarshal(target, buffer, size)                               \
1538        Unmarshal(TPMI_RH_NV_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
```

```
1539  #define TPMI_RH_LOCKOUT_Unmarshal(target, buffer, size)                          \
1540      Unmarshal(TPMI_RH_LOCKOUT_MARSHAL_INDEX, (target), (buffer), (size))
1541  #define TPMI_RH_NV_INDEX_Unmarshal(target, buffer, size)                         \
1542      Unmarshal(TPMI_RH_NV_INDEX_MARSHAL_INDEX, (target), (buffer), (size))
1543  #define TPMI_RH_NV_INDEX_Marshal(source, buffer, size)                           \
1544      Marshal(TPMI_RH_NV_INDEX_MARSHAL_INDEX, (source), (buffer), (size))
1545  #define TPMI_RH_AC_Unmarshal(target, buffer, size)                               \
1546      Unmarshal(TPMI_RH_AC_MARSHAL_INDEX, (target), (buffer), (size))
1547  #define TPMI_ALG_HASH_Unmarshal(target, buffer, size, flag)                      \
1548      Unmarshal(TPMI_ALG_HASH_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1549        (buffer), (size))
1550  #define TPMI_ALG_HASH_Marshal(source, buffer, size)                              \
1551      Marshal(TPMI_ALG_HASH_MARSHAL_INDEX, (source), (buffer), (size))
1552  #define TPMI_ALG_ASYM_Unmarshal(target, buffer, size, flag)                      \
1553      Unmarshal(TPMI_ALG_ASYM_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1554        (buffer), (size))
1555  #define TPMI_ALG_ASYM_Marshal(source, buffer, size)                              \
1556      Marshal(TPMI_ALG_ASYM_MARSHAL_INDEX, (source), (buffer), (size))
1557  #define TPMI_ALG_SYM_Unmarshal(target, buffer, size, flag)                       \
1558      Unmarshal(TPMI_ALG_SYM_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
1559        (buffer), (size))
1560  #define TPMI_ALG_SYM_Marshal(source, buffer, size)                               \
1561      Marshal(TPMI_ALG_SYM_MARSHAL_INDEX, (source), (buffer), (size))
1562  #define TPMI_ALG_SYM_OBJECT_Unmarshal(target, buffer, size, flag)                \
1563      Unmarshal(TPMI_ALG_SYM_OBJECT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1564        (buffer), (size))
1565  #define TPMI_ALG_SYM_OBJECT_Marshal(source, buffer, size)                        \
1566      Marshal(TPMI_ALG_SYM_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1567  #define TPMI_ALG_SYM_MODE_Unmarshal(target, buffer, size, flag)                  \
1568      Unmarshal(TPMI_ALG_SYM_MODE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1569        (buffer), (size))
1570  #define TPMI_ALG_SYM_MODE_Marshal(source, buffer, size)                          \
1571      Marshal(TPMI_ALG_SYM_MODE_MARSHAL_INDEX, (source), (buffer), (size))
1572  #define TPMI_ALG_KDF_Unmarshal(target, buffer, size, flag)                       \
1573      Unmarshal(TPMI_ALG_KDF_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
1574        (buffer), (size))
1575  #define TPMI_ALG_KDF_Marshal(source, buffer, size)                               \
1576      Marshal(TPMI_ALG_KDF_MARSHAL_INDEX, (source), (buffer), (size))
1577  #define TPMI_ALG_SIG_SCHEME_Unmarshal(target, buffer, size, flag)                \
1578      Unmarshal(TPMI_ALG_SIG_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1579        (buffer), (size))
1580  #define TPMI_ALG_SIG_SCHEME_Marshal(source, buffer, size)                        \
1581      Marshal(TPMI_ALG_SIG_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1582  #define TPMI_ECC_KEY_EXCHANGE_Unmarshal(target, buffer, size, flag)              \
1583      Unmarshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0),     \
1584        (target), (buffer), (size))
1585  #define TPMI_ECC_KEY_EXCHANGE_Marshal(source, buffer, size)                      \
1586      Marshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_INDEX, (source), (buffer), (size))
1587  #define TPMI_ST_COMMAND_TAG_Unmarshal(target, buffer, size)                      \
1588      Unmarshal(TPMI_ST_COMMAND_TAG_MARSHAL_INDEX, (target), (buffer), (size))
1589  #define TPMI_ST_COMMAND_TAG_Marshal(source, buffer, size)                        \
1590      Marshal(TPMI_ST_COMMAND_TAG_MARSHAL_INDEX, (source), (buffer), (size))
1591  #define TPMI_ALG_MAC_SCHEME_Unmarshal(target, buffer, size, flag)                \
1592      Unmarshal(TPMI_ALG_MAC_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1593        (buffer), (size))
1594  #define TPMI_ALG_MAC_SCHEME_Marshal(source, buffer, size)                        \
1595      Marshal(TPMI_ALG_MAC_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1596  #define TPMI_ALG_CIPHER_MODE_Unmarshal(target, buffer, size, flag)               \
1597      Unmarshal(TPMI_ALG_CIPHER_MODE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1598        (buffer), (size))
1599  #define TPMI_ALG_CIPHER_MODE_Marshal(source, buffer, size)                       \
1600      Marshal(TPMI_ALG_CIPHER_MODE_MARSHAL_INDEX, (source), (buffer), (size))
1601  #define TPMS_EMPTY_Unmarshal(target, buffer, size)                               \
1602      Unmarshal(TPMS_EMPTY_MARSHAL_INDEX, (target), (buffer), (size))
1603  #define TPMS_EMPTY_Marshal(source, buffer, size)                                 \
1604      Marshal(TPMS_EMPTY_MARSHAL_INDEX, (source), (buffer), (size))
```

```
1605  #define TPMS_ALGORITHM_DESCRIPTION_Marshal(source, buffer, size)                    \
1606      Marshal(TPMS_ALGORITHM_DESCRIPTION_MARSHAL_INDEX, (source), (buffer), (size))
1607  #define TPMU_HA_Unmarshal(target, buffer, size, selector)                           \
1608      UnmarshalUnion(TPMU_HA_MARSHAL_INDEX, (target), (buffer), (size), (selector))
1609  #define TPMU_HA_Marshal(source, buffer, size, selector)                             \
1610      MarshalUnion(TPMU_HA_MARSHAL_INDEX, (target), (buffer), (size), (selector))
1611  #define TPMT_HA_Unmarshal(target, buffer, size, flag)                               \
1612      Unmarshal(TPMT_HA_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), (buffer),   \
1613        (size))
1614  #define TPMT_HA_Marshal(source, buffer, size)                                       \
1615      Marshal(TPMT_HA_MARSHAL_INDEX, (source), (buffer), (size))
1616  #define TPM2B_DIGEST_Unmarshal(target, buffer, size)                                \
1617      Unmarshal(TPM2B_DIGEST_MARSHAL_INDEX, (target), (buffer), (size))
1618  #define TPM2B_DIGEST_Marshal(source, buffer, size)                                  \
1619      Marshal(TPM2B_DIGEST_MARSHAL_INDEX, (source), (buffer), (size))
1620  #define TPM2B_DATA_Unmarshal(target, buffer, size)                                  \
1621      Unmarshal(TPM2B_DATA_MARSHAL_INDEX, (target), (buffer), (size))
1622  #define TPM2B_DATA_Marshal(source, buffer, size)                                    \
1623      Marshal(TPM2B_DATA_MARSHAL_INDEX, (source), (buffer), (size))
1624  #define TPM2B_NONCE_Unmarshal(target, buffer, size)                                 \
1625      Unmarshal(TPM2B_NONCE_MARSHAL_INDEX, (target), (buffer), (size))
1626  #define TPM2B_NONCE_Marshal(source, buffer, size)                                   \
1627      Marshal(TPM2B_NONCE_MARSHAL_INDEX, (source), (buffer), (size))
1628  #define TPM2B_AUTH_Unmarshal(target, buffer, size)                                  \
1629      Unmarshal(TPM2B_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
1630  #define TPM2B_AUTH_Marshal(source, buffer, size)                                    \
1631      Marshal(TPM2B_AUTH_MARSHAL_INDEX, (source), (buffer), (size))
1632  #define TPM2B_OPERAND_Unmarshal(target, buffer, size)                               \
1633      Unmarshal(TPM2B_OPERAND_MARSHAL_INDEX, (target), (buffer), (size))
1634  #define TPM2B_OPERAND_Marshal(source, buffer, size)                                 \
1635      Marshal(TPM2B_OPERAND_MARSHAL_INDEX, (source), (buffer), (size))
1636  #define TPM2B_EVENT_Unmarshal(target, buffer, size)                                 \
1637      Unmarshal(TPM2B_EVENT_MARSHAL_INDEX, (target), (buffer), (size))
1638  #define TPM2B_EVENT_Marshal(source, buffer, size)                                   \
1639      Marshal(TPM2B_EVENT_MARSHAL_INDEX, (source), (buffer), (size))
1640  #define TPM2B_MAX_BUFFER_Unmarshal(target, buffer, size)                            \
1641      Unmarshal(TPM2B_MAX_BUFFER_MARSHAL_INDEX, (target), (buffer), (size))
1642  #define TPM2B_MAX_BUFFER_Marshal(source, buffer, size)                              \
1643      Marshal(TPM2B_MAX_BUFFER_MARSHAL_INDEX, (source), (buffer), (size))
1644  #define TPM2B_MAX_NV_BUFFER_Unmarshal(target, buffer, size)                         \
1645      Unmarshal(TPM2B_MAX_NV_BUFFER_MARSHAL_INDEX, (target), (buffer), (size))
1646  #define TPM2B_MAX_NV_BUFFER_Marshal(source, buffer, size)                           \
1647      Marshal(TPM2B_MAX_NV_BUFFER_MARSHAL_INDEX, (source), (buffer), (size))
1648  #define TPM2B_TIMEOUT_Unmarshal(target, buffer, size)                               \
1649      Unmarshal(TPM2B_TIMEOUT_MARSHAL_INDEX, (target), (buffer), (size))
1650  #define TPM2B_TIMEOUT_Marshal(source, buffer, size)                                 \
1651      Marshal(TPM2B_TIMEOUT_MARSHAL_INDEX, (source), (buffer), (size))
1652  #define TPM2B_IV_Unmarshal(target, buffer, size)                                    \
1653      Unmarshal(TPM2B_IV_MARSHAL_INDEX, (target), (buffer), (size))
1654  #define TPM2B_IV_Marshal(source, buffer, size)                                      \
1655      Marshal(TPM2B_IV_MARSHAL_INDEX, (source), (buffer), (size))
1656  #define TPM2B_NAME_Unmarshal(target, buffer, size)                                  \
1657      Unmarshal(TPM2B_NAME_MARSHAL_INDEX, (target), (buffer), (size))
1658  #define TPM2B_NAME_Marshal(source, buffer, size)                                    \
1659      Marshal(TPM2B_NAME_MARSHAL_INDEX, (source), (buffer), (size))
1660  #define TPMS_PCR_SELECT_Unmarshal(target, buffer, size)                             \
1661      Unmarshal(TPMS_PCR_SELECT_MARSHAL_INDEX, (target), (buffer), (size))
1662  #define TPMS_PCR_SELECT_Marshal(source, buffer, size)                               \
1663      Marshal(TPMS_PCR_SELECT_MARSHAL_INDEX, (source), (buffer), (size))
1664  #define TPMS_PCR_SELECTION_Unmarshal(target, buffer, size)                          \
1665      Unmarshal(TPMS_PCR_SELECTION_MARSHAL_INDEX, (target), (buffer), (size))
1666  #define TPMS_PCR_SELECTION_Marshal(source, buffer, size)                            \
1667      Marshal(TPMS_PCR_SELECTION_MARSHAL_INDEX, (source), (buffer), (size))
1668  #define TPMT_TK_CREATION_Unmarshal(target, buffer, size)                            \
1669      Unmarshal(TPMT_TK_CREATION_MARSHAL_INDEX, (target), (buffer), (size))
1670  #define TPMT_TK_CREATION_Marshal(source, buffer, size)                              \
```

```
1671        Marshal(TPMT_TK_CREATION_MARSHAL_INDEX, (source), (buffer), (size))
1672    #define TPMT_TK_VERIFIED_Unmarshal(target, buffer, size)                    \
1673        Unmarshal(TPMT_TK_VERIFIED_MARSHAL_INDEX, (target), (buffer), (size))
1674    #define TPMT_TK_VERIFIED_Marshal(source, buffer, size)                      \
1675        Marshal(TPMT_TK_VERIFIED_MARSHAL_INDEX, (source), (buffer), (size))
1676    #define TPMT_TK_AUTH_Unmarshal(target, buffer, size)                        \
1677        Unmarshal(TPMT_TK_AUTH_MARSHAL_INDEX, (target), (buffer), (size))
1678    #define TPMT_TK_AUTH_Marshal(source, buffer, size)                          \
1679        Marshal(TPMT_TK_AUTH_MARSHAL_INDEX, (source), (buffer), (size))
1680    #define TPMT_TK_HASHCHECK_Unmarshal(target, buffer, size)                   \
1681        Unmarshal(TPMT_TK_HASHCHECK_MARSHAL_INDEX, (target), (buffer), (size))
1682    #define TPMT_TK_HASHCHECK_Marshal(source, buffer, size)                     \
1683        Marshal(TPMT_TK_HASHCHECK_MARSHAL_INDEX, (source), (buffer), (size))
1684    #define TPMS_ALG_PROPERTY_Marshal(source, buffer, size)                     \
1685        Marshal(TPMS_ALG_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1686    #define TPMS_TAGGED_PROPERTY_Marshal(source, buffer, size)                  \
1687        Marshal(TPMS_TAGGED_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1688    #define TPMS_TAGGED_PCR_SELECT_Marshal(source, buffer, size)                \
1689        Marshal(TPMS_TAGGED_PCR_SELECT_MARSHAL_INDEX, (source), (buffer), (size))
1690    #define TPMS_TAGGED_POLICY_Marshal(source, buffer, size)                    \
1691        Marshal(TPMS_TAGGED_POLICY_MARSHAL_INDEX, (source), (buffer), (size))
1692    #define TPML_CC_Unmarshal(target, buffer, size)                             \
1693        Unmarshal(TPML_CC_MARSHAL_INDEX, (target), (buffer), (size))
1694    #define TPML_CC_Marshal(source, buffer, size)                               \
1695        Marshal(TPML_CC_MARSHAL_INDEX, (source), (buffer), (size))
1696    #define TPML_CCA_Marshal(source, buffer, size)                              \
1697        Marshal(TPML_CCA_MARSHAL_INDEX, (source), (buffer), (size))
1698    #define TPML_ALG_Unmarshal(target, buffer, size)                            \
1699        Unmarshal(TPML_ALG_MARSHAL_INDEX, (target), (buffer), (size))
1700    #define TPML_ALG_Marshal(source, buffer, size)                              \
1701        Marshal(TPML_ALG_MARSHAL_INDEX, (source), (buffer), (size))
1702    #define TPML_HANDLE_Marshal(source, buffer, size)                           \
1703        Marshal(TPML_HANDLE_MARSHAL_INDEX, (source), (buffer), (size))
1704    #define TPML_DIGEST_Unmarshal(target, buffer, size)                         \
1705        Unmarshal(TPML_DIGEST_MARSHAL_INDEX, (target), (buffer), (size))
1706    #define TPML_DIGEST_Marshal(source, buffer, size)                           \
1707        Marshal(TPML_DIGEST_MARSHAL_INDEX, (source), (buffer), (size))
1708    #define TPML_DIGEST_VALUES_Unmarshal(target, buffer, size)                  \
1709        Unmarshal(TPML_DIGEST_VALUES_MARSHAL_INDEX, (target), (buffer), (size))
1710    #define TPML_DIGEST_VALUES_Marshal(source, buffer, size)                    \
1711        Marshal(TPML_DIGEST_VALUES_MARSHAL_INDEX, (source), (buffer), (size))
1712    #define TPML_PCR_SELECTION_Unmarshal(target, buffer, size)                  \
1713        Unmarshal(TPML_PCR_SELECTION_MARSHAL_INDEX, (target), (buffer), (size))
1714    #define TPML_PCR_SELECTION_Marshal(source, buffer, size)                    \
1715        Marshal(TPML_PCR_SELECTION_MARSHAL_INDEX, (source), (buffer), (size))
1716    #define TPML_ALG_PROPERTY_Marshal(source, buffer, size)                     \
1717        Marshal(TPML_ALG_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1718    #define TPML_TAGGED_TPM_PROPERTY_Marshal(source, buffer, size)              \
1719        Marshal(TPML_TAGGED_TPM_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1720    #define TPML_TAGGED_PCR_PROPERTY_Marshal(source, buffer, size)              \
1721        Marshal(TPML_TAGGED_PCR_PROPERTY_MARSHAL_INDEX, (source), (buffer), (size))
1722    #define TPML_ECC_CURVE_Marshal(source, buffer, size)                        \
1723        Marshal(TPML_ECC_CURVE_MARSHAL_INDEX, (source), (buffer), (size))
1724    #define TPML_TAGGED_POLICY_Marshal(source, buffer, size)                    \
1725        Marshal(TPML_TAGGED_POLICY_MARSHAL_INDEX, (source), (buffer), (size))
1726    #define TPMU_CAPABILITIES_Marshal(source, buffer, size, selector)           \
1727        MarshalUnion(TPMU_CAPABILITIES_MARSHAL_INDEX, (target), (buffer), (size),   \
1728          (selector))
1729    #define TPMS_CAPABILITY_DATA_Marshal(source, buffer, size)                  \
1730        Marshal(TPMS_CAPABILITY_DATA_MARSHAL_INDEX, (source), (buffer), (size))
1731    #define TPMS_CLOCK_INFO_Unmarshal(target, buffer, size)                     \
1732        Unmarshal(TPMS_CLOCK_INFO_MARSHAL_INDEX, (target), (buffer), (size))
1733    #define TPMS_CLOCK_INFO_Marshal(source, buffer, size)                       \
1734        Marshal(TPMS_CLOCK_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1735    #define TPMS_TIME_INFO_Unmarshal(target, buffer, size)                      \
1736        Unmarshal(TPMS_TIME_INFO_MARSHAL_INDEX, (target), (buffer), (size))
```

```
1737  #define TPMS_TIME_INFO_Marshal(source, buffer, size)                                \
1738      Marshal(TPMS_TIME_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1739  #define TPMS_TIME_ATTEST_INFO_Marshal(source, buffer, size)                         \
1740      Marshal(TPMS_TIME_ATTEST_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1741  #define TPMS_CERTIFY_INFO_Marshal(source, buffer, size)                             \
1742      Marshal(TPMS_CERTIFY_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1743  #define TPMS_QUOTE_INFO_Marshal(source, buffer, size)                               \
1744      Marshal(TPMS_QUOTE_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1745  #define TPMS_COMMAND_AUDIT_INFO_Marshal(source, buffer, size)                       \
1746      Marshal(TPMS_COMMAND_AUDIT_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1747  #define TPMS_SESSION_AUDIT_INFO_Marshal(source, buffer, size)                       \
1748      Marshal(TPMS_SESSION_AUDIT_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1749  #define TPMS_CREATION_INFO_Marshal(source, buffer, size)                            \
1750      Marshal(TPMS_CREATION_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1751  #define TPMS_NV_CERTIFY_INFO_Marshal(source, buffer, size)                          \
1752      Marshal(TPMS_NV_CERTIFY_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1753  #define TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(source, buffer, size)                   \
1754      Marshal(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_INDEX, (source), (buffer), (size))
1755  #define TPMI_ST_ATTEST_Marshal(source, buffer, size)                                \
1756      Marshal(TPMI_ST_ATTEST_MARSHAL_INDEX, (source), (buffer), (size))
1757  #define TPMU_ATTEST_Marshal(source, buffer, size, selector)                         \
1758      MarshalUnion(TPMU_ATTEST_MARSHAL_INDEX, (target), (buffer), (size), (selector))
1759  #define TPMS_ATTEST_Marshal(source, buffer, size)                                   \
1760      Marshal(TPMS_ATTEST_MARSHAL_INDEX, (source), (buffer), (size))
1761  #define TPM2B_ATTEST_Marshal(source, buffer, size)                                  \
1762      Marshal(TPM2B_ATTEST_MARSHAL_INDEX, (source), (buffer), (size))
1763  #define TPMS_AUTH_COMMAND_Unmarshal(target, buffer, size)                           \
1764      Unmarshal(TPMS_AUTH_COMMAND_MARSHAL_INDEX, (target), (buffer), (size))
1765  #define TPMS_AUTH_RESPONSE_Marshal(source, buffer, size)                            \
1766      Marshal(TPMS_AUTH_RESPONSE_MARSHAL_INDEX, (source), (buffer), (size))
1767  #define TPMI_TDES_KEY_BITS_Unmarshal(target, buffer, size)                          \
1768      Unmarshal(TPMI_TDES_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1769  #define TPMI_TDES_KEY_BITS_Marshal(source, buffer, size)                            \
1770      Marshal(TPMI_TDES_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1771  #define TPMI_AES_KEY_BITS_Unmarshal(target, buffer, size)                           \
1772      Unmarshal(TPMI_AES_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1773  #define TPMI_AES_KEY_BITS_Marshal(source, buffer, size)                             \
1774      Marshal(TPMI_AES_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1775  #define TPMI_SM4_KEY_BITS_Unmarshal(target, buffer, size)                           \
1776      Unmarshal(TPMI_SM4_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1777  #define TPMI_SM4_KEY_BITS_Marshal(source, buffer, size)                             \
1778      Marshal(TPMI_SM4_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1779  #define TPMI_CAMELLIA_KEY_BITS_Unmarshal(target, buffer, size)                      \
1780      Unmarshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1781  #define TPMI_CAMELLIA_KEY_BITS_Marshal(source, buffer, size)                        \
1782      Marshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1783  #define TPMU_SYM_KEY_BITS_Unmarshal(target, buffer, size, selector)                 \
1784      UnmarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size),     \
1785        (selector))
1786  #define TPMU_SYM_KEY_BITS_Marshal(source, buffer, size, selector)                   \
1787      MarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size),       \
1788        (selector))
1789  #define TPMU_SYM_MODE_Unmarshal(target, buffer, size, selector)                     \
1790      UnmarshalUnion(TPMU_SYM_MODE_MARSHAL_INDEX, (target), (buffer), (size),         \
1791        (selector))
1792  #define TPMU_SYM_MODE_Marshal(source, buffer, size, selector)                       \
1793      MarshalUnion(TPMU_SYM_MODE_MARSHAL_INDEX, (target), (buffer), (size),           \
1794        (selector))
1795  #define TPMT_SYM_DEF_Unmarshal(target, buffer, size, flag)                          \
1796      Unmarshal(TPMT_SYM_DEF_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),       \
1797        (buffer), (size))
1798  #define TPMT_SYM_DEF_Marshal(source, buffer, size)                                  \
1799      Marshal(TPMT_SYM_DEF_MARSHAL_INDEX, (source), (buffer), (size))
1800  #define TPMT_SYM_DEF_OBJECT_Unmarshal(target, buffer, size, flag)                   \
1801      Unmarshal(TPMT_SYM_DEF_OBJECT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1802        (buffer), (size))
```

```
1803  #define TPMT_SYM_DEF_OBJECT_Marshal(source, buffer, size)                          \
1804      Marshal(TPMT_SYM_DEF_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
1805  #define TPM2B_SYM_KEY_Unmarshal(target, buffer, size)                              \
1806      Unmarshal(TPM2B_SYM_KEY_MARSHAL_INDEX, (target), (buffer), (size))
1807  #define TPM2B_SYM_KEY_Marshal(source, buffer, size)                                \
1808      Marshal(TPM2B_SYM_KEY_MARSHAL_INDEX, (source), (buffer), (size))
1809  #define TPMS_SYMCIPHER_PARMS_Unmarshal(target, buffer, size)                       \
1810      Unmarshal(TPMS_SYMCIPHER_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
1811  #define TPMS_SYMCIPHER_PARMS_Marshal(source, buffer, size)                         \
1812      Marshal(TPMS_SYMCIPHER_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
1813  #define TPM2B_LABEL_Unmarshal(target, buffer, size)                                \
1814      Unmarshal(TPM2B_LABEL_MARSHAL_INDEX, (target), (buffer), (size))
1815  #define TPM2B_LABEL_Marshal(source, buffer, size)                                  \
1816      Marshal(TPM2B_LABEL_MARSHAL_INDEX, (source), (buffer), (size))
1817  #define TPMS_DERIVE_Unmarshal(target, buffer, size)                                \
1818      Unmarshal(TPMS_DERIVE_MARSHAL_INDEX, (target), (buffer), (size))
1819  #define TPMS_DERIVE_Marshal(source, buffer, size)                                  \
1820      Marshal(TPMS_DERIVE_MARSHAL_INDEX, (source), (buffer), (size))
1821  #define TPM2B_DERIVE_Unmarshal(target, buffer, size)                               \
1822      Unmarshal(TPM2B_DERIVE_MARSHAL_INDEX, (target), (buffer), (size))
1823  #define TPM2B_DERIVE_Marshal(source, buffer, size)                                 \
1824      Marshal(TPM2B_DERIVE_MARSHAL_INDEX, (source), (buffer), (size))
1825  #define TPM2B_SENSITIVE_DATA_Unmarshal(target, buffer, size)                       \
1826      Unmarshal(TPM2B_SENSITIVE_DATA_MARSHAL_INDEX, (target), (buffer), (size))
1827  #define TPM2B_SENSITIVE_DATA_Marshal(source, buffer, size)                         \
1828      Marshal(TPM2B_SENSITIVE_DATA_MARSHAL_INDEX, (source), (buffer), (size))
1829  #define TPMS_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                      \
1830      Unmarshal(TPMS_SENSITIVE_CREATE_MARSHAL_INDEX, (target), (buffer), (size))
1831  #define TPM2B_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                     \
1832      Unmarshal(TPM2B_SENSITIVE_CREATE_MARSHAL_INDEX, (target), (buffer), (size))
1833  #define TPMS_SCHEME_HASH_Unmarshal(target, buffer, size)                           \
1834      Unmarshal(TPMS_SCHEME_HASH_MARSHAL_INDEX, (target), (buffer), (size))
1835  #define TPMS_SCHEME_HASH_Marshal(source, buffer, size)                             \
1836      Marshal(TPMS_SCHEME_HASH_MARSHAL_INDEX, (source), (buffer), (size))
1837  #define TPMS_SCHEME_ECDAA_Unmarshal(target, buffer, size)                          \
1838      Unmarshal(TPMS_SCHEME_ECDAA_MARSHAL_INDEX, (target), (buffer), (size))
1839  #define TPMS_SCHEME_ECDAA_Marshal(source, buffer, size)                            \
1840      Marshal(TPMS_SCHEME_ECDAA_MARSHAL_INDEX, (source), (buffer), (size))
1841  #define TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)            \
1842      Unmarshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0),    \
1843        (target), (buffer), (size))
1844  #define TPMI_ALG_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                    \
1845      Marshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1846  #define TPMS_SCHEME_HMAC_Unmarshal(target, buffer, size)                           \
1847      Unmarshal(TPMS_SCHEME_HMAC_MARSHAL_INDEX, (target), (buffer), (size))
1848  #define TPMS_SCHEME_HMAC_Marshal(source, buffer, size)                             \
1849      Marshal(TPMS_SCHEME_HMAC_MARSHAL_INDEX, (source), (buffer), (size))
1850  #define TPMS_SCHEME_XOR_Unmarshal(target, buffer, size)                            \
1851      Unmarshal(TPMS_SCHEME_XOR_MARSHAL_INDEX, (target), (buffer), (size))
1852  #define TPMS_SCHEME_XOR_Marshal(source, buffer, size)                              \
1853      Marshal(TPMS_SCHEME_XOR_MARSHAL_INDEX, (source), (buffer), (size))
1854  #define TPMU_SCHEME_KEYEDHASH_Unmarshal(target, buffer, size, selector)            \
1855      UnmarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_INDEX, (target), (buffer), (size), \
1856        (selector))
1857  #define TPMU_SCHEME_KEYEDHASH_Marshal(source, buffer, size, selector)              \
1858      MarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_INDEX, (target), (buffer), (size),  \
1859        (selector))
1860  #define TPMT_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)                \
1861      Unmarshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0),        \
1862        (target), (buffer), (size))
1863  #define TPMT_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                        \
1864      Marshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1865  #define TPMS_SIG_SCHEME_RSASSA_Unmarshal(target, buffer, size)                     \
1866      Unmarshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_INDEX, (target), (buffer), (size))
1867  #define TPMS_SIG_SCHEME_RSASSA_Marshal(source, buffer, size)                       \
1868      Marshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_INDEX, (source), (buffer), (size))
```

```
1869  #define TPMS_SIG_SCHEME_RSAPSS_Unmarshal(target, buffer, size)                    \
1870      Unmarshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_INDEX, (target), (buffer), (size))
1871  #define TPMS_SIG_SCHEME_RSAPSS_Marshal(source, buffer, size)                      \
1872      Marshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_INDEX, (source), (buffer), (size))
1873  #define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size)                     \
1874      Unmarshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_INDEX, (target), (buffer), (size))
1875  #define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size)                       \
1876      Marshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_INDEX, (source), (buffer), (size))
1877  #define TPMS_SIG_SCHEME_SM2_Unmarshal(target, buffer, size)                       \
1878      Unmarshal(TPMS_SIG_SCHEME_SM2_MARSHAL_INDEX, (target), (buffer), (size))
1879  #define TPMS_SIG_SCHEME_SM2_Marshal(source, buffer, size)                         \
1880      Marshal(TPMS_SIG_SCHEME_SM2_MARSHAL_INDEX, (source), (buffer), (size))
1881  #define TPMS_SIG_SCHEME_ECSCHNORR_Unmarshal(target, buffer, size)                 \
1882      Unmarshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_INDEX, (target), (buffer), (size))
1883  #define TPMS_SIG_SCHEME_ECSCHNORR_Marshal(source, buffer, size)                   \
1884      Marshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_INDEX, (source), (buffer), (size))
1885  #define TPMS_SIG_SCHEME_ECDAA_Unmarshal(target, buffer, size)                     \
1886      Unmarshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_INDEX, (target), (buffer), (size))
1887  #define TPMS_SIG_SCHEME_ECDAA_Marshal(source, buffer, size)                       \
1888      Marshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_INDEX, (source), (buffer), (size))
1889  #define TPMU_SIG_SCHEME_Unmarshal(target, buffer, size, selector)                 \
1890      UnmarshalUnion(TPMU_SIG_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),     \
1891        (selector))
1892  #define TPMU_SIG_SCHEME_Marshal(source, buffer, size, selector)                   \
1893      MarshalUnion(TPMU_SIG_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),       \
1894        (selector))
1895  #define TPMT_SIG_SCHEME_Unmarshal(target, buffer, size, flag)                     \
1896      Unmarshal(TPMT_SIG_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
1897        (buffer), (size))
1898  #define TPMT_SIG_SCHEME_Marshal(source, buffer, size)                             \
1899      Marshal(TPMT_SIG_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1900  #define TPMS_ENC_SCHEME_OAEP_Unmarshal(target, buffer, size)                      \
1901      Unmarshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_INDEX, (target), (buffer), (size))
1902  #define TPMS_ENC_SCHEME_OAEP_Marshal(source, buffer, size)                        \
1903      Marshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_INDEX, (source), (buffer), (size))
1904  #define TPMS_ENC_SCHEME_RSAES_Unmarshal(target, buffer, size)                     \
1905      Unmarshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_INDEX, (target), (buffer), (size))
1906  #define TPMS_ENC_SCHEME_RSAES_Marshal(source, buffer, size)                       \
1907      Marshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_INDEX, (source), (buffer), (size))
1908  #define TPMS_KEY_SCHEME_ECDH_Unmarshal(target, buffer, size)                      \
1909      Unmarshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_INDEX, (target), (buffer), (size))
1910  #define TPMS_KEY_SCHEME_ECDH_Marshal(source, buffer, size)                        \
1911      Marshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_INDEX, (source), (buffer), (size))
1912  #define TPMS_KEY_SCHEME_ECMQV_Unmarshal(target, buffer, size)                     \
1913      Unmarshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_INDEX, (target), (buffer), (size))
1914  #define TPMS_KEY_SCHEME_ECMQV_Marshal(source, buffer, size)                       \
1915      Marshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_INDEX, (source), (buffer), (size))
1916  #define TPMS_SCHEME_MGF1_Unmarshal(target, buffer, size)                          \
1917      Unmarshal(TPMS_SCHEME_MGF1_MARSHAL_INDEX, (target), (buffer), (size))
1918  #define TPMS_SCHEME_MGF1_Marshal(source, buffer, size)                            \
1919      Marshal(TPMS_SCHEME_MGF1_MARSHAL_INDEX, (source), (buffer), (size))
1920  #define TPMS_SCHEME_KDF1_SP800_56A_Unmarshal(target, buffer, size)                \
1921      Unmarshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_INDEX, (target), (buffer), (size))
1922  #define TPMS_SCHEME_KDF1_SP800_56A_Marshal(source, buffer, size)                  \
1923      Marshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_INDEX, (source), (buffer), (size))
1924  #define TPMS_SCHEME_KDF2_Unmarshal(target, buffer, size)                          \
1925      Unmarshal(TPMS_SCHEME_KDF2_MARSHAL_INDEX, (target), (buffer), (size))
1926  #define TPMS_SCHEME_KDF2_Marshal(source, buffer, size)                            \
1927      Marshal(TPMS_SCHEME_KDF2_MARSHAL_INDEX, (source), (buffer), (size))
1928  #define TPMS_SCHEME_KDF1_SP800_108_Unmarshal(target, buffer, size)                \
1929      Unmarshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_INDEX, (target), (buffer), (size))
1930  #define TPMS_SCHEME_KDF1_SP800_108_Marshal(source, buffer, size)                  \
1931      Marshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_INDEX, (source), (buffer), (size))
1932  #define TPMU_KDF_SCHEME_Unmarshal(target, buffer, size, selector)                 \
1933      UnmarshalUnion(TPMU_KDF_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),     \
1934        (selector))
```

```
1935  #define TPMU_KDF_SCHEME_Marshal(source, buffer, size, selector)                  \
1936      MarshalUnion(TPMU_KDF_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),      \
1937        (selector))
1938  #define TPMT_KDF_SCHEME_Unmarshal(target, buffer, size, flag)                    \
1939      Unmarshal(TPMT_KDF_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),  \
1940        (buffer), (size))
1941  #define TPMT_KDF_SCHEME_Marshal(source, buffer, size)                            \
1942      Marshal(TPMT_KDF_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1943  #define TPMI_ALG_ASYM_SCHEME_Unmarshal(target, buffer, size, flag)               \
1944      Unmarshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1945        (buffer), (size))
1946  #define TPMI_ALG_ASYM_SCHEME_Marshal(source, buffer, size)                       \
1947      Marshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1948  #define TPMU_ASYM_SCHEME_Unmarshal(target, buffer, size, selector)               \
1949      UnmarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),   \
1950        (selector))
1951  #define TPMU_ASYM_SCHEME_Marshal(source, buffer, size, selector)                 \
1952      MarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_INDEX, (target), (buffer), (size),     \
1953        (selector))
1954  #define TPMI_ALG_RSA_SCHEME_Unmarshal(target, buffer, size, flag)                \
1955      Unmarshal(TPMI_ALG_RSA_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
1956        (buffer), (size))
1957  #define TPMI_ALG_RSA_SCHEME_Marshal(source, buffer, size)                        \
1958      Marshal(TPMI_ALG_RSA_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1959  #define TPMT_RSA_SCHEME_Unmarshal(target, buffer, size, flag)                    \
1960      Unmarshal(TPMT_RSA_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),  \
1961        (buffer), (size))
1962  #define TPMT_RSA_SCHEME_Marshal(source, buffer, size)                            \
1963      Marshal(TPMT_RSA_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
1964  #define TPMI_ALG_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)               \
1965      Unmarshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),\
1966        (buffer), (size))
1967  #define TPMI_ALG_RSA_DECRYPT_Marshal(source, buffer, size)                       \
1968      Marshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_INDEX, (source), (buffer), (size))
1969  #define TPMT_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)                   \
1970      Unmarshal(TPMT_RSA_DECRYPT_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),  \
1971        (buffer), (size))
1972  #define TPMT_RSA_DECRYPT_Marshal(source, buffer, size)                           \
1973      Marshal(TPMT_RSA_DECRYPT_MARSHAL_INDEX, (source), (buffer), (size))
1974  #define TPM2B_PUBLIC_KEY_RSA_Unmarshal(target, buffer, size)                     \
1975      Unmarshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_INDEX, (target), (buffer), (size))
1976  #define TPM2B_PUBLIC_KEY_RSA_Marshal(source, buffer, size)                       \
1977      Marshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_INDEX, (source), (buffer), (size))
1978  #define TPMI_RSA_KEY_BITS_Unmarshal(target, buffer, size)                        \
1979      Unmarshal(TPMI_RSA_KEY_BITS_MARSHAL_INDEX, (target), (buffer), (size))
1980  #define TPMI_RSA_KEY_BITS_Marshal(source, buffer, size)                          \
1981      Marshal(TPMI_RSA_KEY_BITS_MARSHAL_INDEX, (source), (buffer), (size))
1982  #define TPM2B_PRIVATE_KEY_RSA_Unmarshal(target, buffer, size)                    \
1983      Unmarshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_INDEX, (target), (buffer), (size))
1984  #define TPM2B_PRIVATE_KEY_RSA_Marshal(source, buffer, size)                      \
1985      Marshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_INDEX, (source), (buffer), (size))
1986  #define TPM2B_ECC_PARAMETER_Unmarshal(target, buffer, size)                      \
1987      Unmarshal(TPM2B_ECC_PARAMETER_MARSHAL_INDEX, (target), (buffer), (size))
1988  #define TPM2B_ECC_PARAMETER_Marshal(source, buffer, size)                        \
1989      Marshal(TPM2B_ECC_PARAMETER_MARSHAL_INDEX, (source), (buffer), (size))
1990  #define TPMS_ECC_POINT_Unmarshal(target, buffer, size)                           \
1991      Unmarshal(TPMS_ECC_POINT_MARSHAL_INDEX, (target), (buffer), (size))
1992  #define TPMS_ECC_POINT_Marshal(source, buffer, size)                             \
1993      Marshal(TPMS_ECC_POINT_MARSHAL_INDEX, (source), (buffer), (size))
1994  #define TPM2B_ECC_POINT_Unmarshal(target, buffer, size)                          \
1995      Unmarshal(TPM2B_ECC_POINT_MARSHAL_INDEX, (target), (buffer), (size))
1996  #define TPM2B_ECC_POINT_Marshal(source, buffer, size)                            \
1997      Marshal(TPM2B_ECC_POINT_MARSHAL_INDEX, (source), (buffer), (size))
1998  #define TPMI_ALG_ECC_SCHEME_Unmarshal(target, buffer, size, flag)                \
1999      Unmarshal(TPMI_ALG_ECC_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target), \
2000        (buffer), (size))
```

```
2001 #define TPMI_ALG_ECC_SCHEME_Marshal(source, buffer, size)                          \
2002     Marshal(TPMI_ALG_ECC_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
2003 #define TPMI_ECC_CURVE_Unmarshal(target, buffer, size)                             \
2004     Unmarshal(TPMI_ECC_CURVE_MARSHAL_INDEX, (target), (buffer), (size))
2005 #define TPMI_ECC_CURVE_Marshal(source, buffer, size)                               \
2006     Marshal(TPMI_ECC_CURVE_MARSHAL_INDEX, (source), (buffer), (size))
2007 #define TPMT_ECC_SCHEME_Unmarshal(target, buffer, size, flag)                      \
2008     Unmarshal(TPMT_ECC_SCHEME_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),   \
2009       (buffer), (size))
2010 #define TPMT_ECC_SCHEME_Marshal(source, buffer, size)                              \
2011     Marshal(TPMT_ECC_SCHEME_MARSHAL_INDEX, (source), (buffer), (size))
2012 #define TPMS_ALGORITHM_DETAIL_ECC_Marshal(source, buffer, size)                    \
2013     Marshal(TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_INDEX, (source), (buffer), (size))
2014 #define TPMS_SIGNATURE_RSA_Unmarshal(target, buffer, size)                         \
2015     Unmarshal(TPMS_SIGNATURE_RSA_MARSHAL_INDEX, (target), (buffer), (size))
2016 #define TPMS_SIGNATURE_RSA_Marshal(source, buffer, size)                           \
2017     Marshal(TPMS_SIGNATURE_RSA_MARSHAL_INDEX, (source), (buffer), (size))
2018 #define TPMS_SIGNATURE_RSASSA_Unmarshal(target, buffer, size)                      \
2019     Unmarshal(TPMS_SIGNATURE_RSASSA_MARSHAL_INDEX, (target), (buffer), (size))
2020 #define TPMS_SIGNATURE_RSASSA_Marshal(source, buffer, size)                        \
2021     Marshal(TPMS_SIGNATURE_RSASSA_MARSHAL_INDEX, (source), (buffer), (size))
2022 #define TPMS_SIGNATURE_RSAPSS_Unmarshal(target, buffer, size)                      \
2023     Unmarshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_INDEX, (target), (buffer), (size))
2024 #define TPMS_SIGNATURE_RSAPSS_Marshal(source, buffer, size)                        \
2025     Marshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_INDEX, (source), (buffer), (size))
2026 #define TPMS_SIGNATURE_ECC_Unmarshal(target, buffer, size)                         \
2027     Unmarshal(TPMS_SIGNATURE_ECC_MARSHAL_INDEX, (target), (buffer), (size))
2028 #define TPMS_SIGNATURE_ECC_Marshal(source, buffer, size)                           \
2029     Marshal(TPMS_SIGNATURE_ECC_MARSHAL_INDEX, (source), (buffer), (size))
2030 #define TPMS_SIGNATURE_ECDAA_Unmarshal(target, buffer, size)                       \
2031     Unmarshal(TPMS_SIGNATURE_ECDAA_MARSHAL_INDEX, (target), (buffer), (size))
2032 #define TPMS_SIGNATURE_ECDAA_Marshal(source, buffer, size)                         \
2033     Marshal(TPMS_SIGNATURE_ECDAA_MARSHAL_INDEX, (source), (buffer), (size))
2034 #define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size)                       \
2035     Unmarshal(TPMS_SIGNATURE_ECDSA_MARSHAL_INDEX, (target), (buffer), (size))
2036 #define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size)                         \
2037     Marshal(TPMS_SIGNATURE_ECDSA_MARSHAL_INDEX, (source), (buffer), (size))
2038 #define TPMS_SIGNATURE_SM2_Unmarshal(target, buffer, size)                         \
2039     Unmarshal(TPMS_SIGNATURE_SM2_MARSHAL_INDEX, (target), (buffer), (size))
2040 #define TPMS_SIGNATURE_SM2_Marshal(source, buffer, size)                           \
2041     Marshal(TPMS_SIGNATURE_SM2_MARSHAL_INDEX, (source), (buffer), (size))
2042 #define TPMS_SIGNATURE_ECSCHNORR_Unmarshal(target, buffer, size)                   \
2043     Unmarshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_INDEX, (target), (buffer), (size))
2044 #define TPMS_SIGNATURE_ECSCHNORR_Marshal(source, buffer, size)                     \
2045     Marshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_INDEX, (source), (buffer), (size))
2046 #define TPMU_SIGNATURE_Unmarshal(target, buffer, size, selector)                   \
2047     UnmarshalUnion(TPMU_SIGNATURE_MARSHAL_INDEX, (target), (buffer), (size),      \
2048       (selector))
2049 #define TPMU_SIGNATURE_Marshal(source, buffer, size, selector)                     \
2050     MarshalUnion(TPMU_SIGNATURE_MARSHAL_INDEX, (target), (buffer), (size),        \
2051       (selector))
2052 #define TPMT_SIGNATURE_Unmarshal(target, buffer, size, flag)                       \
2053     Unmarshal(TPMT_SIGNATURE_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
2054       (buffer), (size))
2055 #define TPMT_SIGNATURE_Marshal(source, buffer, size)                               \
2056     Marshal(TPMT_SIGNATURE_MARSHAL_INDEX, (source), (buffer), (size))
2057 #define TPMU_ENCRYPTED_SECRET_Unmarshal(target, buffer, size, selector)            \
2058     UnmarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_INDEX, (target), (buffer), (size), \
2059       (selector))
2060 #define TPMU_ENCRYPTED_SECRET_Marshal(source, buffer, size, selector)              \
2061     MarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_INDEX, (target), (buffer), (size),  \
2062       (selector))
2063 #define TPM2B_ENCRYPTED_SECRET_Unmarshal(target, buffer, size)                     \
2064     Unmarshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_INDEX, (target), (buffer), (size))
2065 #define TPM2B_ENCRYPTED_SECRET_Marshal(source, buffer, size)                       \
2066     Marshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_INDEX, (source), (buffer), (size))
```

```
2067  #define TPMI_ALG_PUBLIC_Unmarshal(target, buffer, size)                          \
2068      Unmarshal(TPMI_ALG_PUBLIC_MARSHAL_INDEX, (target), (buffer), (size))
2069  #define TPMI_ALG_PUBLIC_Marshal(source, buffer, size)                            \
2070      Marshal(TPMI_ALG_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2071  #define TPMU_PUBLIC_ID_Unmarshal(target, buffer, size, selector)                 \
2072      UnmarshalUnion(TPMU_PUBLIC_ID_MARSHAL_INDEX, (target), (buffer), (size),     \
2073        (selector))
2074  #define TPMU_PUBLIC_ID_Marshal(source, buffer, size, selector)                   \
2075      MarshalUnion(TPMU_PUBLIC_ID_MARSHAL_INDEX, (target), (buffer), (size),       \
2076        (selector))
2077  #define TPMS_KEYEDHASH_PARMS_Unmarshal(target, buffer, size)                     \
2078      Unmarshal(TPMS_KEYEDHASH_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2079  #define TPMS_KEYEDHASH_PARMS_Marshal(source, buffer, size)                       \
2080      Marshal(TPMS_KEYEDHASH_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2081  #define TPMS_RSA_PARMS_Unmarshal(target, buffer, size)                           \
2082      Unmarshal(TPMS_RSA_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2083  #define TPMS_RSA_PARMS_Marshal(source, buffer, size)                             \
2084      Marshal(TPMS_RSA_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2085  #define TPMS_ECC_PARMS_Unmarshal(target, buffer, size)                           \
2086      Unmarshal(TPMS_ECC_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2087  #define TPMS_ECC_PARMS_Marshal(source, buffer, size)                             \
2088      Marshal(TPMS_ECC_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2089  #define TPMU_PUBLIC_PARMS_Unmarshal(target, buffer, size, selector)              \
2090      UnmarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_INDEX, (target), (buffer), (size),  \
2091        (selector))
2092  #define TPMU_PUBLIC_PARMS_Marshal(source, buffer, size, selector)                \
2093      MarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_INDEX, (target), (buffer), (size),    \
2094        (selector))
2095  #define TPMT_PUBLIC_PARMS_Unmarshal(target, buffer, size)                        \
2096      Unmarshal(TPMT_PUBLIC_PARMS_MARSHAL_INDEX, (target), (buffer), (size))
2097  #define TPMT_PUBLIC_PARMS_Marshal(source, buffer, size)                          \
2098      Marshal(TPMT_PUBLIC_PARMS_MARSHAL_INDEX, (source), (buffer), (size))
2099  #define TPMT_PUBLIC_Unmarshal(target, buffer, size, flag)                        \
2100      Unmarshal(TPMT_PUBLIC_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),     \
2101        (buffer), (size))
2102  #define TPMT_PUBLIC_Marshal(source, buffer, size)                                \
2103      Marshal(TPMT_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2104  #define TPM2B_PUBLIC_Unmarshal(target, buffer, size, flag)                       \
2105      Unmarshal(TPM2B_PUBLIC_MARSHAL_INDEX | (flag ? NULL_FLAG : 0), (target),    \
2106        (buffer), (size))
2107  #define TPM2B_PUBLIC_Marshal(source, buffer, size)                               \
2108      Marshal(TPM2B_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2109  #define TPM2B_TEMPLATE_Unmarshal(target, buffer, size)                           \
2110      Unmarshal(TPM2B_TEMPLATE_MARSHAL_INDEX, (target), (buffer), (size))
2111  #define TPM2B_TEMPLATE_Marshal(source, buffer, size)                             \
2112      Marshal(TPM2B_TEMPLATE_MARSHAL_INDEX, (source), (buffer), (size))
2113  #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(target, buffer, size)            \
2114      Unmarshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_INDEX, (target), (buffer),   \
2115        (size))
2116  #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(source, buffer, size)              \
2117      Marshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_INDEX, (source), (buffer), (size))
2118  #define TPMU_SENSITIVE_COMPOSITE_Unmarshal(target, buffer, size, selector)       \
2119      UnmarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_INDEX, (target), (buffer),   \
2120        (size), (selector))
2121  #define TPMU_SENSITIVE_COMPOSITE_Marshal(source, buffer, size, selector)         \
2122      MarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_INDEX, (target), (buffer), (size),\
2123        (selector))
2124  #define TPMT_SENSITIVE_Unmarshal(target, buffer, size)                           \
2125      Unmarshal(TPMT_SENSITIVE_MARSHAL_INDEX, (target), (buffer), (size))
2126  #define TPMT_SENSITIVE_Marshal(source, buffer, size)                             \
2127      Marshal(TPMT_SENSITIVE_MARSHAL_INDEX, (source), (buffer), (size))
2128  #define TPM2B_SENSITIVE_Unmarshal(target, buffer, size)                          \
2129      Unmarshal(TPM2B_SENSITIVE_MARSHAL_INDEX, (target), (buffer), (size))
2130  #define TPM2B_SENSITIVE_Marshal(source, buffer, size)                            \
2131      Marshal(TPM2B_SENSITIVE_MARSHAL_INDEX, (source), (buffer), (size))
2132  #define TPM2B_PRIVATE_Unmarshal(target, buffer, size)                            \
```

```
2133        Unmarshal(TPM2B_PRIVATE_MARSHAL_INDEX, (target), (buffer), (size))
2134    #define TPM2B_PRIVATE_Marshal(source, buffer, size)                                 \
2135        Marshal(TPM2B_PRIVATE_MARSHAL_INDEX, (source), (buffer), (size))
2136    #define TPM2B_ID_OBJECT_Unmarshal(target, buffer, size)                             \
2137        Unmarshal(TPM2B_ID_OBJECT_MARSHAL_INDEX, (target), (buffer), (size))
2138    #define TPM2B_ID_OBJECT_Marshal(source, buffer, size)                               \
2139        Marshal(TPM2B_ID_OBJECT_MARSHAL_INDEX, (source), (buffer), (size))
2140    #define TPM_NV_INDEX_Marshal(source, buffer, size)                                  \
2141        Marshal(TPM_NV_INDEX_MARSHAL_INDEX, (source), (buffer), (size))
2142    #define TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(target, buffer, size)              \
2143        Unmarshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_INDEX, (target), (buffer),     \
2144          (size))
2145    #define TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(source, buffer, size)                \
2146        Marshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_INDEX, (source), (buffer),       \
2147          (size))
2148    #define TPMA_NV_Unmarshal(target, buffer, size)                                     \
2149        Unmarshal(TPMA_NV_MARSHAL_INDEX, (target), (buffer), (size))
2150    #define TPMA_NV_Marshal(source, buffer, size)                                       \
2151        Marshal(TPMA_NV_MARSHAL_INDEX, (source), (buffer), (size))
2152    #define TPMS_NV_PUBLIC_Unmarshal(target, buffer, size)                              \
2153        Unmarshal(TPMS_NV_PUBLIC_MARSHAL_INDEX, (target), (buffer), (size))
2154    #define TPMS_NV_PUBLIC_Marshal(source, buffer, size)                                \
2155        Marshal(TPMS_NV_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2156    #define TPM2B_NV_PUBLIC_Unmarshal(target, buffer, size)                             \
2157        Unmarshal(TPM2B_NV_PUBLIC_MARSHAL_INDEX, (target), (buffer), (size))
2158    #define TPM2B_NV_PUBLIC_Marshal(source, buffer, size)                               \
2159        Marshal(TPM2B_NV_PUBLIC_MARSHAL_INDEX, (source), (buffer), (size))
2160    #define TPM2B_CONTEXT_SENSITIVE_Unmarshal(target, buffer, size)                     \
2161        Unmarshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_INDEX, (target), (buffer), (size))
2162    #define TPM2B_CONTEXT_SENSITIVE_Marshal(source, buffer, size)                       \
2163        Marshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_INDEX, (source), (buffer), (size))
2164    #define TPMS_CONTEXT_DATA_Unmarshal(target, buffer, size)                           \
2165        Unmarshal(TPMS_CONTEXT_DATA_MARSHAL_INDEX, (target), (buffer), (size))
2166    #define TPMS_CONTEXT_DATA_Marshal(source, buffer, size)                             \
2167        Marshal(TPMS_CONTEXT_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2168    #define TPM2B_CONTEXT_DATA_Unmarshal(target, buffer, size)                          \
2169        Unmarshal(TPM2B_CONTEXT_DATA_MARSHAL_INDEX, (target), (buffer), (size))
2170    #define TPM2B_CONTEXT_DATA_Marshal(source, buffer, size)                            \
2171        Marshal(TPM2B_CONTEXT_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2172    #define TPMS_CONTEXT_Unmarshal(target, buffer, size)                                \
2173        Unmarshal(TPMS_CONTEXT_MARSHAL_INDEX, (target), (buffer), (size))
2174    #define TPMS_CONTEXT_Marshal(source, buffer, size)                                  \
2175        Marshal(TPMS_CONTEXT_MARSHAL_INDEX, (source), (buffer), (size))
2176    #define TPMS_CREATION_DATA_Marshal(source, buffer, size)                            \
2177        Marshal(TPMS_CREATION_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2178    #define TPM2B_CREATION_DATA_Marshal(source, buffer, size)                           \
2179        Marshal(TPM2B_CREATION_DATA_MARSHAL_INDEX, (source), (buffer), (size))
2180    #define TPM_AT_Unmarshal(target, buffer, size)                                      \
2181        Unmarshal(TPM_AT_MARSHAL_INDEX, (target), (buffer), (size))
2182    #define TPM_AT_Marshal(source, buffer, size)                                        \
2183        Marshal(TPM_AT_MARSHAL_INDEX, (source), (buffer), (size))
2184    #define TPM_AE_Marshal(source, buffer, size)                                        \
2185        Marshal(TPM_AE_MARSHAL_INDEX, (source), (buffer), (size))
2186    #define TPMS_AC_OUTPUT_Marshal(source, buffer, size)                                \
2187        Marshal(TPMS_AC_OUTPUT_MARSHAL_INDEX, (source), (buffer), (size))
2188    #define TPML_AC_CAPABILITIES_Marshal(source, buffer, size)                          \
2189        Marshal(TPML_AC_CAPABILITIES_MARSHAL_INDEX, (source), (buffer), (size))
2190    #endif // _Table_Marshal_Data_
```

9.10.7.3 TableMarshalDefines.h

```
1    #ifndef _TABLE_MARSHAL_DEFINES_H_
2    #define _TABLE_MARSHAL_DEFINES_H_
3    #define NULL_SHIFT  15
4    #define NULL_FLAG   (1 << NULL_SHIFT)
```

The range macro processes a min, max value and produces a values that is used in the computation to see if something is within a range. The max value is (max-min). This lets the check for something (*val*) within a range become: if((val - min) <= max) // passes if in range if((val - min) > max) // passes if not in range This works because all values are converted to UINT32 values before the compare. For (val - min), all values greater than or equal to val will become positive values with a value equal to *min* being zero. This means that in an unsigned compare against 'max,' any value that is outside the range will appear to be a number greater than max. The benefit of this operation is that this will work even if the input value is a signed number as long as the input is sign extended.

```
5   #define RANGE(_min_, _max_, _base_)                                    \
6       (UINT32)_min_, (UINT32)((_base_)(_max_ - _min_))
```

This macro is like the offsetof macro but, instead of computing the offset of a structure element, it computes the stride between elements that are in a structure array. This is used instead of sizeof() because the sizeof() operator on a structure can return an implementation dependent value.

```
7   #define STRIDE(s)    ((UINT16)(size_t)&(((s *)0)[1]))
8   #define MARSHAL_REF(TYPE)    ((UINT16)(offsetof(MARSHAL_DATA, TYPE)))
```

This macro creates the entry in the array lookup table

```
9   #define ARRAY_MARSHAL_ENTRY(TYPE)                                      \
10          {(marshalIndex_t)TYPE##_MARSHAL_REF, (UINT16)STRIDE(TYPE)}
```

Defines for array lookup

```
11  #define UINT8_ARRAY_MARSHAL_INDEX                   0   // 0x00
12  #define TPM_CC_ARRAY_MARSHAL_INDEX                  1   // 0x01
13  #define TPMA_CC_ARRAY_MARSHAL_INDEX                 2   // 0x02
14  #define TPM_ALG_ID_ARRAY_MARSHAL_INDEX              3   // 0x03
15  #define TPM_HANDLE_ARRAY_MARSHAL_INDEX              4   // 0x04
16  #define TPM2B_DIGEST_ARRAY_MARSHAL_INDEX            5   // 0x05
17  #define TPMT_HA_ARRAY_MARSHAL_INDEX                 6   // 0x06
18  #define TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX      7   // 0x07
19  #define TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX       8   // 0x08
20  #define TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX    9   // 0x09
21  #define TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX  10  // 0x0A
22  #define TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX           11  // 0x0B
23  #define TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX      12  // 0x0C
24  #define TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX           13  // 0x0D
25  #define TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX          14  // 0x0E
```

Defines for referencing a type by offset

```
26  #define UINT8_MARSHAL_REF                           \
27              ((UINT16)(offsetof(MarshalData_st, UINT8_DATA)))
28  #define BYTE_MARSHAL_REF                            UINT8_MARSHAL_REF
29  #define TPM_HT_MARSHAL_REF                          UINT8_MARSHAL_REF
30  #define TPMA_LOCALITY_MARSHAL_REF                   UINT8_MARSHAL_REF
31  #define UINT16_MARSHAL_REF                          \
32              ((UINT16)(offsetof(MarshalData_st, UINT16_DATA)))
33  #define TPM_KEY_SIZE_MARSHAL_REF                    UINT16_MARSHAL_REF
34  #define TPM_KEY_BITS_MARSHAL_REF                    UINT16_MARSHAL_REF
35  #define TPM_ALG_ID_MARSHAL_REF                      UINT16_MARSHAL_REF
36  #define TPM_ST_MARSHAL_REF                          UINT16_MARSHAL_REF
37  #define UINT32_MARSHAL_REF                          \
38              ((UINT16)(offsetof(MarshalData_st, UINT32_DATA)))
39  #define TPM_ALGORITHM_ID_MARSHAL_REF                UINT32_MARSHAL_REF
40  #define TPM_MODIFIER_INDICATOR_MARSHAL_REF          UINT32_MARSHAL_REF
41  #define TPM_AUTHORIZATION_SIZE_MARSHAL_REF          UINT32_MARSHAL_REF
42  #define TPM_PARAMETER_SIZE_MARSHAL_REF              UINT32_MARSHAL_REF
43  #define TPM_SPEC_MARSHAL_REF                        UINT32_MARSHAL_REF
```



```
#define TPM_GENERATED_MARSHAL_REF                   UINT32_MARSHAL_REF
#define TPM_CC_MARSHAL_REF                          UINT32_MARSHAL_REF
#define TPM_RC_MARSHAL_REF                          UINT32_MARSHAL_REF
#define TPM_PT_MARSHAL_REF                          UINT32_MARSHAL_REF
#define TPM_PT_PCR_MARSHAL_REF                      UINT32_MARSHAL_REF
#define TPM_PS_MARSHAL_REF                          UINT32_MARSHAL_REF
#define TPM_HANDLE_MARSHAL_REF                      UINT32_MARSHAL_REF
#define TPM_RH_MARSHAL_REF                          UINT32_MARSHAL_REF
#define TPM_HC_MARSHAL_REF                          UINT32_MARSHAL_REF
#define TPMA_PERMANENT_MARSHAL_REF                  UINT32_MARSHAL_REF
#define TPMA_STARTUP_CLEAR_MARSHAL_REF              UINT32_MARSHAL_REF
#define TPMA_MEMORY_MARSHAL_REF                     UINT32_MARSHAL_REF
#define TPMA_CC_MARSHAL_REF                         UINT32_MARSHAL_REF
#define TPMA_MODES_MARSHAL_REF                      UINT32_MARSHAL_REF
#define TPMA_X509_KEY_USAGE_MARSHAL_REF             UINT32_MARSHAL_REF
#define TPM_NV_INDEX_MARSHAL_REF                    UINT32_MARSHAL_REF
#define TPM_AE_MARSHAL_REF                          UINT32_MARSHAL_REF
#define UINT64_MARSHAL_REF                          \
    ((UINT16)(offsetof(MarshalData_st, UINT64_DATA)))
#define INT8_MARSHAL_REF                            \
    ((UINT16)(offsetof(MarshalData_st, INT8_DATA)))
#define INT16_MARSHAL_REF                           \
    ((UINT16)(offsetof(MarshalData_st, INT16_DATA)))
#define INT32_MARSHAL_REF                           \
    ((UINT16)(offsetof(MarshalData_st, INT32_DATA)))
#define INT64_MARSHAL_REF                           \
    ((UINT16)(offsetof(MarshalData_st, INT64_DATA)))
#define UINT0_MARSHAL_REF                           \
    ((UINT16)(offsetof(MarshalData_st, UINT0_DATA)))
#define TPM_ECC_CURVE_MARSHAL_REF                   \
    ((UINT16)(offsetof(MarshalData_st, TPM_ECC_CURVE_DATA)))
#define TPM_CLOCK_ADJUST_MARSHAL_REF                \
    ((UINT16)(offsetof(MarshalData_st, TPM_CLOCK_ADJUST_DATA)))
#define TPM_EO_MARSHAL_REF                          \
    ((UINT16)(offsetof(MarshalData_st, TPM_EO_DATA)))
#define TPM_SU_MARSHAL_REF                          \
    ((UINT16)(offsetof(MarshalData_st, TPM_SU_DATA)))
#define TPM_SE_MARSHAL_REF                          \
    ((UINT16)(offsetof(MarshalData_st, TPM_SE_DATA)))
#define TPM_CAP_MARSHAL_REF                         \
    ((UINT16)(offsetof(MarshalData_st, TPM_CAP_DATA)))
#define TPMA_ALGORITHM_MARSHAL_REF                  \
    ((UINT16)(offsetof(MarshalData_st, TPMA_ALGORITHM_DATA)))
#define TPMA_OBJECT_MARSHAL_REF                     \
    ((UINT16)(offsetof(MarshalData_st, TPMA_OBJECT_DATA)))
#define TPMA_SESSION_MARSHAL_REF                    \
    ((UINT16)(offsetof(MarshalData_st, TPMA_SESSION_DATA)))
#define TPMA_ACT_MARSHAL_REF                        \
    ((UINT16)(offsetof(MarshalData_st, TPMA_ACT_DATA)))
#define TPMI_YES_NO_MARSHAL_REF                     \
    ((UINT16)(offsetof(MarshalData_st, TPMI_YES_NO_DATA)))
#define TPMI_DH_OBJECT_MARSHAL_REF                  \
    ((UINT16)(offsetof(MarshalData_st, TPMI_DH_OBJECT_DATA)))
#define TPMI_DH_PARENT_MARSHAL_REF                  \
    ((UINT16)(offsetof(MarshalData_st, TPMI_DH_PARENT_DATA)))
#define TPMI_DH_PERSISTENT_MARSHAL_REF              \
    ((UINT16)(offsetof(MarshalData_st, TPMI_DH_PERSISTENT_DATA)))
#define TPMI_DH_ENTITY_MARSHAL_REF                  \
    ((UINT16)(offsetof(MarshalData_st, TPMI_DH_ENTITY_DATA)))
#define TPMI_DH_PCR_MARSHAL_REF                     \
    ((UINT16)(offsetof(MarshalData_st, TPMI_DH_PCR_DATA)))
#define TPMI_SH_AUTH_SESSION_MARSHAL_REF            \
    ((UINT16)(offsetof(MarshalData_st, TPMI_SH_AUTH_SESSION_DATA)))
#define TPMI_SH_HMAC_MARSHAL_REF                    \
    ((UINT16)(offsetof(MarshalData_st, TPMI_SH_HMAC_DATA)))
#define TPMI_SH_POLICY_MARSHAL_REF                  \
```

```
110             ((UINT16)(offsetof(MarshalData_st, TPMI_SH_POLICY_DATA)))
111 #define TPMI_DH_CONTEXT_MARSHAL_REF                     \
112             ((UINT16)(offsetof(MarshalData_st, TPMI_DH_CONTEXT_DATA)))
113 #define TPMI_DH_SAVED_MARSHAL_REF                       \
114             ((UINT16)(offsetof(MarshalData_st, TPMI_DH_SAVED_DATA)))
115 #define TPMI_RH_HIERARCHY_MARSHAL_REF                   \
116             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_HIERARCHY_DATA)))
117 #define TPMI_RH_ENABLES_MARSHAL_REF                     \
118             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_ENABLES_DATA)))
119 #define TPMI_RH_HIERARCHY_AUTH_MARSHAL_REF              \
120             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_HIERARCHY_AUTH_DATA)))
121 #define TPMI_RH_HIERARCHY_POLICY_MARSHAL_REF            \
122             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_HIERARCHY_POLICY_DATA)))
123 #define TPMI_RH_PLATFORM_MARSHAL_REF                    \
124             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_PLATFORM_DATA)))
125 #define TPMI_RH_OWNER_MARSHAL_REF                       \
126             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_OWNER_DATA)))
127 #define TPMI_RH_ENDORSEMENT_MARSHAL_REF                 \
128             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_ENDORSEMENT_DATA)))
129 #define TPMI_RH_PROVISION_MARSHAL_REF                   \
130             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_PROVISION_DATA)))
131 #define TPMI_RH_CLEAR_MARSHAL_REF                       \
132             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_CLEAR_DATA)))
133 #define TPMI_RH_NV_AUTH_MARSHAL_REF                     \
134             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_NV_AUTH_DATA)))
135 #define TPMI_RH_LOCKOUT_MARSHAL_REF                     \
136             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_LOCKOUT_DATA)))
137 #define TPMI_RH_NV_INDEX_MARSHAL_REF                    \
138             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_NV_INDEX_DATA)))
139 #define TPMI_RH_AC_MARSHAL_REF                          \
140             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_AC_DATA)))
141 #define TPMI_RH_ACT_MARSHAL_REF                         \
142             ((UINT16)(offsetof(MarshalData_st, TPMI_RH_ACT_DATA)))
143 #define TPMI_ALG_HASH_MARSHAL_REF                       \
144             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_HASH_DATA)))
145 #define TPMI_ALG_ASYM_MARSHAL_REF                       \
146             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_ASYM_DATA)))
147 #define TPMI_ALG_SYM_MARSHAL_REF                        \
148             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SYM_DATA)))
149 #define TPMI_ALG_SYM_OBJECT_MARSHAL_REF                 \
150             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SYM_OBJECT_DATA)))
151 #define TPMI_ALG_SYM_MODE_MARSHAL_REF                   \
152             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SYM_MODE_DATA)))
153 #define TPMI_ALG_KDF_MARSHAL_REF                        \
154             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_KDF_DATA)))
155 #define TPMI_ALG_SIG_SCHEME_MARSHAL_REF                 \
156             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_SIG_SCHEME_DATA)))
157 #define TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF               \
158             ((UINT16)(offsetof(MarshalData_st, TPMI_ECC_KEY_EXCHANGE_DATA)))
159 #define TPMI_ST_COMMAND_TAG_MARSHAL_REF                 \
160             ((UINT16)(offsetof(MarshalData_st, TPMI_ST_COMMAND_TAG_DATA)))
161 #define TPMI_ALG_MAC_SCHEME_MARSHAL_REF                 \
162             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_MAC_SCHEME_DATA)))
163 #define TPMI_ALG_CIPHER_MODE_MARSHAL_REF                \
164             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_CIPHER_MODE_DATA)))
165 #define TPMS_EMPTY_MARSHAL_REF                          \
166             ((UINT16)(offsetof(MarshalData_st, TPMS_EMPTY_DATA)))
167 #define TPMS_ENC_SCHEME_RSAES_MARSHAL_REF           TPMS_EMPTY_MARSHAL_REF
168 #define TPMS_ALGORITHM_DESCRIPTION_MARSHAL_REF      \
169             ((UINT16)(offsetof(MarshalData_st, TPMS_ALGORITHM_DESCRIPTION_DATA)))
170 #define TPMU_HA_MARSHAL_REF                             \
171             ((UINT16)(offsetof(MarshalData_st, TPMU_HA_DATA)))
172 #define TPMT_HA_MARSHAL_REF                             \
173             ((UINT16)(offsetof(MarshalData_st, TPMT_HA_DATA)))
174 #define TPM2B_DIGEST_MARSHAL_REF                        \
175             ((UINT16)(offsetof(MarshalData_st, TPM2B_DIGEST_DATA)))
```

```
176  #define TPM2B_NONCE_MARSHAL_REF                    TPM2B_DIGEST_MARSHAL_REF
177  #define TPM2B_AUTH_MARSHAL_REF                     TPM2B_DIGEST_MARSHAL_REF
178  #define TPM2B_OPERAND_MARSHAL_REF                  TPM2B_DIGEST_MARSHAL_REF
179  #define TPM2B_DATA_MARSHAL_REF                     \
180              ((UINT16)(offsetof(MarshalData_st, TPM2B_DATA_DATA)))
181  #define TPM2B_EVENT_MARSHAL_REF                    \
182              ((UINT16)(offsetof(MarshalData_st, TPM2B_EVENT_DATA)))
183  #define TPM2B_MAX_BUFFER_MARSHAL_REF               \
184              ((UINT16)(offsetof(MarshalData_st, TPM2B_MAX_BUFFER_DATA)))
185  #define TPM2B_MAX_NV_BUFFER_MARSHAL_REF            \
186              ((UINT16)(offsetof(MarshalData_st, TPM2B_MAX_NV_BUFFER_DATA)))
187  #define TPM2B_TIMEOUT_MARSHAL_REF                  \
188              ((UINT16)(offsetof(MarshalData_st, TPM2B_TIMEOUT_DATA)))
189  #define TPM2B_IV_MARSHAL_REF                       \
190              ((UINT16)(offsetof(MarshalData_st, TPM2B_IV_DATA)))
191  #define NULL_UNION_MARSHAL_REF                     \
192              ((UINT16)(offsetof(MarshalData_st, NULL_UNION_DATA)))
193  #define TPMU_NAME_MARSHAL_REF                      NULL_UNION_MARSHAL_REF
194  #define TPMU_SENSITIVE_CREATE_MARSHAL_REF          NULL_UNION_MARSHAL_REF
195  #define TPM2B_NAME_MARSHAL_REF                     \
196              ((UINT16)(offsetof(MarshalData_st, TPM2B_NAME_DATA)))
197  #define TPMS_PCR_SELECT_MARSHAL_REF                \
198              ((UINT16)(offsetof(MarshalData_st, TPMS_PCR_SELECT_DATA)))
199  #define TPMS_PCR_SELECTION_MARSHAL_REF             \
200              ((UINT16)(offsetof(MarshalData_st, TPMS_PCR_SELECTION_DATA)))
201  #define TPMT_TK_CREATION_MARSHAL_REF               \
202              ((UINT16)(offsetof(MarshalData_st, TPMT_TK_CREATION_DATA)))
203  #define TPMT_TK_VERIFIED_MARSHAL_REF               \
204              ((UINT16)(offsetof(MarshalData_st, TPMT_TK_VERIFIED_DATA)))
205  #define TPMT_TK_AUTH_MARSHAL_REF                   \
206              ((UINT16)(offsetof(MarshalData_st, TPMT_TK_AUTH_DATA)))
207  #define TPMT_TK_HASHCHECK_MARSHAL_REF              \
208              ((UINT16)(offsetof(MarshalData_st, TPMT_TK_HASHCHECK_DATA)))
209  #define TPMS_ALG_PROPERTY_MARSHAL_REF              \
210              ((UINT16)(offsetof(MarshalData_st, TPMS_ALG_PROPERTY_DATA)))
211  #define TPMS_TAGGED_PROPERTY_MARSHAL_REF           \
212              ((UINT16)(offsetof(MarshalData_st, TPMS_TAGGED_PROPERTY_DATA)))
213  #define TPMS_TAGGED_PCR_SELECT_MARSHAL_REF         \
214              ((UINT16)(offsetof(MarshalData_st, TPMS_TAGGED_PCR_SELECT_DATA)))
215  #define TPMS_TAGGED_POLICY_MARSHAL_REF             \
216              ((UINT16)(offsetof(MarshalData_st, TPMS_TAGGED_POLICY_DATA)))
217  #define TPMS_ACT_DATA_MARSHAL_REF                  \
218              ((UINT16)(offsetof(MarshalData_st, TPMS_ACT_DATA_DATA)))
219  #define TPML_CC_MARSHAL_REF                        \
220              ((UINT16)(offsetof(MarshalData_st, TPML_CC_DATA)))
221  #define TPML_CCA_MARSHAL_REF                       \
222              ((UINT16)(offsetof(MarshalData_st, TPML_CCA_DATA)))
223  #define TPML_ALG_MARSHAL_REF                       \
224              ((UINT16)(offsetof(MarshalData_st, TPML_ALG_DATA)))
225  #define TPML_HANDLE_MARSHAL_REF                    \
226              ((UINT16)(offsetof(MarshalData_st, TPML_HANDLE_DATA)))
227  #define TPML_DIGEST_MARSHAL_REF                    \
228              ((UINT16)(offsetof(MarshalData_st, TPML_DIGEST_DATA)))
229  #define TPML_DIGEST_VALUES_MARSHAL_REF             \
230              ((UINT16)(offsetof(MarshalData_st, TPML_DIGEST_VALUES_DATA)))
231  #define TPML_PCR_SELECTION_MARSHAL_REF             \
232              ((UINT16)(offsetof(MarshalData_st, TPML_PCR_SELECTION_DATA)))
233  #define TPML_ALG_PROPERTY_MARSHAL_REF              \
234              ((UINT16)(offsetof(MarshalData_st, TPML_ALG_PROPERTY_DATA)))
235  #define TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF       \
236              ((UINT16)(offsetof(MarshalData_st, TPML_TAGGED_TPM_PROPERTY_DATA)))
237  #define TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF       \
238              ((UINT16)(offsetof(MarshalData_st, TPML_TAGGED_PCR_PROPERTY_DATA)))
239  #define TPML_ECC_CURVE_MARSHAL_REF                 \
240              ((UINT16)(offsetof(MarshalData_st, TPML_ECC_CURVE_DATA)))
241  #define TPML_TAGGED_POLICY_MARSHAL_REF             \
```

```
242             ((UINT16)(offsetof(MarshalData_st, TPML_TAGGED_POLICY_DATA)))
243 #define TPML_ACT_DATA_MARSHAL_REF                       \
244             ((UINT16)(offsetof(MarshalData_st, TPML_ACT_DATA_DATA)))
245 #define TPMU_CAPABILITIES_MARSHAL_REF                   \
246             ((UINT16)(offsetof(MarshalData_st, TPMU_CAPABILITIES_DATA)))
247 #define TPMS_CAPABILITY_DATA_MARSHAL_REF                \
248             ((UINT16)(offsetof(MarshalData_st, TPMS_CAPABILITY_DATA_DATA)))
249 #define TPMS_CLOCK_INFO_MARSHAL_REF                     \
250             ((UINT16)(offsetof(MarshalData_st, TPMS_CLOCK_INFO_DATA)))
251 #define TPMS_TIME_INFO_MARSHAL_REF                      \
252             ((UINT16)(offsetof(MarshalData_st, TPMS_TIME_INFO_DATA)))
253 #define TPMS_TIME_ATTEST_INFO_MARSHAL_REF               \
254             ((UINT16)(offsetof(MarshalData_st, TPMS_TIME_ATTEST_INFO_DATA)))
255 #define TPMS_CERTIFY_INFO_MARSHAL_REF                   \
256             ((UINT16)(offsetof(MarshalData_st, TPMS_CERTIFY_INFO_DATA)))
257 #define TPMS_QUOTE_INFO_MARSHAL_REF                     \
258             ((UINT16)(offsetof(MarshalData_st, TPMS_QUOTE_INFO_DATA)))
259 #define TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF             \
260             ((UINT16)(offsetof(MarshalData_st, TPMS_COMMAND_AUDIT_INFO_DATA)))
261 #define TPMS_SESSION_AUDIT_INFO_MARSHAL_REF             \
262             ((UINT16)(offsetof(MarshalData_st, TPMS_SESSION_AUDIT_INFO_DATA)))
263 #define TPMS_CREATION_INFO_MARSHAL_REF                  \
264             ((UINT16)(offsetof(MarshalData_st, TPMS_CREATION_INFO_DATA)))
265 #define TPMS_NV_CERTIFY_INFO_MARSHAL_REF                \
266             ((UINT16)(offsetof(MarshalData_st, TPMS_NV_CERTIFY_INFO_DATA)))
267 #define TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF         \
268             ((UINT16)(offsetof(MarshalData_st, TPMS_NV_DIGEST_CERTIFY_INFO_DATA)))
269 #define TPMI_ST_ATTEST_MARSHAL_REF                      \
270             ((UINT16)(offsetof(MarshalData_st, TPMI_ST_ATTEST_DATA)))
271 #define TPMU_ATTEST_MARSHAL_REF                         \
272             ((UINT16)(offsetof(MarshalData_st, TPMU_ATTEST_DATA)))
273 #define TPMS_ATTEST_MARSHAL_REF                         \
274             ((UINT16)(offsetof(MarshalData_st, TPMS_ATTEST_DATA)))
275 #define TPM2B_ATTEST_MARSHAL_REF                        \
276             ((UINT16)(offsetof(MarshalData_st, TPM2B_ATTEST_DATA)))
277 #define TPMS_AUTH_COMMAND_MARSHAL_REF                   \
278             ((UINT16)(offsetof(MarshalData_st, TPMS_AUTH_COMMAND_DATA)))
279 #define TPMS_AUTH_RESPONSE_MARSHAL_REF                  \
280             ((UINT16)(offsetof(MarshalData_st, TPMS_AUTH_RESPONSE_DATA)))
281 #define TPMI_TDES_KEY_BITS_MARSHAL_REF                  \
282             ((UINT16)(offsetof(MarshalData_st, TPMI_TDES_KEY_BITS_DATA)))
283 #define TPMI_AES_KEY_BITS_MARSHAL_REF                   \
284             ((UINT16)(offsetof(MarshalData_st, TPMI_AES_KEY_BITS_DATA)))
285 #define TPMI_SM4_KEY_BITS_MARSHAL_REF                   \
286             ((UINT16)(offsetof(MarshalData_st, TPMI_SM4_KEY_BITS_DATA)))
287 #define TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF              \
288             ((UINT16)(offsetof(MarshalData_st, TPMI_CAMELLIA_KEY_BITS_DATA)))
289 #define TPMU_SYM_KEY_BITS_MARSHAL_REF                   \
290             ((UINT16)(offsetof(MarshalData_st, TPMU_SYM_KEY_BITS_DATA)))
291 #define TPMU_SYM_MODE_MARSHAL_REF                       \
292             ((UINT16)(offsetof(MarshalData_st, TPMU_SYM_MODE_DATA)))
293 #define TPMT_SYM_DEF_MARSHAL_REF                        \
294             ((UINT16)(offsetof(MarshalData_st, TPMT_SYM_DEF_DATA)))
295 #define TPMT_SYM_DEF_OBJECT_MARSHAL_REF                 \
296             ((UINT16)(offsetof(MarshalData_st, TPMT_SYM_DEF_OBJECT_DATA)))
297 #define TPM2B_SYM_KEY_MARSHAL_REF                       \
298             ((UINT16)(offsetof(MarshalData_st, TPM2B_SYM_KEY_DATA)))
299 #define TPMS_SYMCIPHER_PARMS_MARSHAL_REF                \
300             ((UINT16)(offsetof(MarshalData_st, TPMS_SYMCIPHER_PARMS_DATA)))
301 #define TPM2B_LABEL_MARSHAL_REF                         \
302             ((UINT16)(offsetof(MarshalData_st, TPM2B_LABEL_DATA)))
303 #define TPMS_DERIVE_MARSHAL_REF                         \
304             ((UINT16)(offsetof(MarshalData_st, TPMS_DERIVE_DATA)))
305 #define TPM2B_DERIVE_MARSHAL_REF                        \
306             ((UINT16)(offsetof(MarshalData_st, TPM2B_DERIVE_DATA)))
307 #define TPM2B_SENSITIVE_DATA_MARSHAL_REF                \
```

```
308             ((UINT16)(offsetof(MarshalData_st, TPM2B_SENSITIVE_DATA_DATA)))
309 #define TPMS_SENSITIVE_CREATE_MARSHAL_REF                \
310             ((UINT16)(offsetof(MarshalData_st, TPMS_SENSITIVE_CREATE_DATA)))
311 #define TPM2B_SENSITIVE_CREATE_MARSHAL_REF               \
312             ((UINT16)(offsetof(MarshalData_st, TPM2B_SENSITIVE_CREATE_DATA)))
313 #define TPMS_SCHEME_HASH_MARSHAL_REF                     \
314             ((UINT16)(offsetof(MarshalData_st, TPMS_SCHEME_HASH_DATA)))
315 #define TPMS_SCHEME_HMAC_MARSHAL_REF                     TPMS_SCHEME_HASH_MARSHAL_REF
316 #define TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF               TPMS_SCHEME_HASH_MARSHAL_REF
317 #define TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF               TPMS_SCHEME_HASH_MARSHAL_REF
318 #define TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF                TPMS_SCHEME_HASH_MARSHAL_REF
319 #define TPMS_SIG_SCHEME_SM2_MARSHAL_REF                  TPMS_SCHEME_HASH_MARSHAL_REF
320 #define TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF            TPMS_SCHEME_HASH_MARSHAL_REF
321 #define TPMS_ENC_SCHEME_OAEP_MARSHAL_REF                 TPMS_SCHEME_HASH_MARSHAL_REF
322 #define TPMS_KEY_SCHEME_ECDH_MARSHAL_REF                 TPMS_SCHEME_HASH_MARSHAL_REF
323 #define TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF                TPMS_SCHEME_HASH_MARSHAL_REF
324 #define TPMS_SCHEME_MGF1_MARSHAL_REF                     TPMS_SCHEME_HASH_MARSHAL_REF
325 #define TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF           TPMS_SCHEME_HASH_MARSHAL_REF
326 #define TPMS_SCHEME_KDF2_MARSHAL_REF                     TPMS_SCHEME_HASH_MARSHAL_REF
327 #define TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF           TPMS_SCHEME_HASH_MARSHAL_REF
328 #define TPMS_SCHEME_ECDAA_MARSHAL_REF                    \
329             ((UINT16)(offsetof(MarshalData_st, TPMS_SCHEME_ECDAA_DATA)))
330 #define TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF                TPMS_SCHEME_ECDAA_MARSHAL_REF
331 #define TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF            \
332             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_KEYEDHASH_SCHEME_DATA)))
333 #define TPMS_SCHEME_XOR_MARSHAL_REF                      \
334             ((UINT16)(offsetof(MarshalData_st, TPMS_SCHEME_XOR_DATA)))
335 #define TPMU_SCHEME_KEYEDHASH_MARSHAL_REF                \
336             ((UINT16)(offsetof(MarshalData_st, TPMU_SCHEME_KEYEDHASH_DATA)))
337 #define TPMT_KEYEDHASH_SCHEME_MARSHAL_REF                \
338             ((UINT16)(offsetof(MarshalData_st, TPMT_KEYEDHASH_SCHEME_DATA)))
339 #define TPMU_SIG_SCHEME_MARSHAL_REF                      \
340             ((UINT16)(offsetof(MarshalData_st, TPMU_SIG_SCHEME_DATA)))
341 #define TPMT_SIG_SCHEME_MARSHAL_REF                      \
342             ((UINT16)(offsetof(MarshalData_st, TPMT_SIG_SCHEME_DATA)))
343 #define TPMU_KDF_SCHEME_MARSHAL_REF                      \
344             ((UINT16)(offsetof(MarshalData_st, TPMU_KDF_SCHEME_DATA)))
345 #define TPMT_KDF_SCHEME_MARSHAL_REF                      \
346             ((UINT16)(offsetof(MarshalData_st, TPMT_KDF_SCHEME_DATA)))
347 #define TPMI_ALG_ASYM_SCHEME_MARSHAL_REF                 \
348             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_ASYM_SCHEME_DATA)))
349 #define TPMU_ASYM_SCHEME_MARSHAL_REF                     \
350             ((UINT16)(offsetof(MarshalData_st, TPMU_ASYM_SCHEME_DATA)))
351 #define TPMI_ALG_RSA_SCHEME_MARSHAL_REF                  \
352             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_RSA_SCHEME_DATA)))
353 #define TPMT_RSA_SCHEME_MARSHAL_REF                      \
354             ((UINT16)(offsetof(MarshalData_st, TPMT_RSA_SCHEME_DATA)))
355 #define TPMI_ALG_RSA_DECRYPT_MARSHAL_REF                 \
356             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_RSA_DECRYPT_DATA)))
357 #define TPMT_RSA_DECRYPT_MARSHAL_REF                     \
358             ((UINT16)(offsetof(MarshalData_st, TPMT_RSA_DECRYPT_DATA)))
359 #define TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF                 \
360             ((UINT16)(offsetof(MarshalData_st, TPM2B_PUBLIC_KEY_RSA_DATA)))
361 #define TPMI_RSA_KEY_BITS_MARSHAL_REF                    \
362             ((UINT16)(offsetof(MarshalData_st, TPMI_RSA_KEY_BITS_DATA)))
363 #define TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF                \
364             ((UINT16)(offsetof(MarshalData_st, TPM2B_PRIVATE_KEY_RSA_DATA)))
365 #define TPM2B_ECC_PARAMETER_MARSHAL_REF                  \
366             ((UINT16)(offsetof(MarshalData_st, TPM2B_ECC_PARAMETER_DATA)))
367 #define TPMS_ECC_POINT_MARSHAL_REF                       \
368             ((UINT16)(offsetof(MarshalData_st, TPMS_ECC_POINT_DATA)))
369 #define TPM2B_ECC_POINT_MARSHAL_REF                      \
370             ((UINT16)(offsetof(MarshalData_st, TPM2B_ECC_POINT_DATA)))
371 #define TPMI_ALG_ECC_SCHEME_MARSHAL_REF                  \
372             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_ECC_SCHEME_DATA)))
373 #define TPMI_ECC_CURVE_MARSHAL_REF                       \
```

```
374             ((UINT16)(offsetof(MarshalData_st, TPMI_ECC_CURVE_DATA)))
375 #define TPMT_ECC_SCHEME_MARSHAL_REF                 \
376             ((UINT16)(offsetof(MarshalData_st, TPMT_ECC_SCHEME_DATA)))
377 #define TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_REF       \
378             ((UINT16)(offsetof(MarshalData_st, TPMS_ALGORITHM_DETAIL_ECC_DATA)))
379 #define TPMS_SIGNATURE_RSA_MARSHAL_REF              \
380             ((UINT16)(offsetof(MarshalData_st, TPMS_SIGNATURE_RSA_DATA)))
381 #define TPMS_SIGNATURE_RSASSA_MARSHAL_REF           TPMS_SIGNATURE_RSA_MARSHAL_REF
382 #define TPMS_SIGNATURE_RSAPSS_MARSHAL_REF           TPMS_SIGNATURE_RSA_MARSHAL_REF
383 #define TPMS_SIGNATURE_ECC_MARSHAL_REF              \
384             ((UINT16)(offsetof(MarshalData_st, TPMS_SIGNATURE_ECC_DATA)))
385 #define TPMS_SIGNATURE_ECDAA_MARSHAL_REF            TPMS_SIGNATURE_ECC_MARSHAL_REF
386 #define TPMS_SIGNATURE_ECDSA_MARSHAL_REF            TPMS_SIGNATURE_ECC_MARSHAL_REF
387 #define TPMS_SIGNATURE_SM2_MARSHAL_REF              TPMS_SIGNATURE_ECC_MARSHAL_REF
388 #define TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF        TPMS_SIGNATURE_ECC_MARSHAL_REF
389 #define TPMU_SIGNATURE_MARSHAL_REF                  \
390             ((UINT16)(offsetof(MarshalData_st, TPMU_SIGNATURE_DATA)))
391 #define TPMT_SIGNATURE_MARSHAL_REF                  \
392             ((UINT16)(offsetof(MarshalData_st, TPMT_SIGNATURE_DATA)))
393 #define TPMU_ENCRYPTED_SECRET_MARSHAL_REF           \
394             ((UINT16)(offsetof(MarshalData_st, TPMU_ENCRYPTED_SECRET_DATA)))
395 #define TPM2B_ENCRYPTED_SECRET_MARSHAL_REF          \
396             ((UINT16)(offsetof(MarshalData_st, TPM2B_ENCRYPTED_SECRET_DATA)))
397 #define TPMI_ALG_PUBLIC_MARSHAL_REF                 \
398             ((UINT16)(offsetof(MarshalData_st, TPMI_ALG_PUBLIC_DATA)))
399 #define TPMU_PUBLIC_ID_MARSHAL_REF                  \
400             ((UINT16)(offsetof(MarshalData_st, TPMU_PUBLIC_ID_DATA)))
401 #define TPMS_KEYEDHASH_PARMS_MARSHAL_REF            \
402             ((UINT16)(offsetof(MarshalData_st, TPMS_KEYEDHASH_PARMS_DATA)))
403 #define TPMS_RSA_PARMS_MARSHAL_REF                  \
404             ((UINT16)(offsetof(MarshalData_st, TPMS_RSA_PARMS_DATA)))
405 #define TPMS_ECC_PARMS_MARSHAL_REF                  \
406             ((UINT16)(offsetof(MarshalData_st, TPMS_ECC_PARMS_DATA)))
407 #define TPMU_PUBLIC_PARMS_MARSHAL_REF               \
408             ((UINT16)(offsetof(MarshalData_st, TPMU_PUBLIC_PARMS_DATA)))
409 #define TPMT_PUBLIC_PARMS_MARSHAL_REF               \
410             ((UINT16)(offsetof(MarshalData_st, TPMT_PUBLIC_PARMS_DATA)))
411 #define TPMT_PUBLIC_MARSHAL_REF                     \
412             ((UINT16)(offsetof(MarshalData_st, TPMT_PUBLIC_DATA)))
413 #define TPM2B_PUBLIC_MARSHAL_REF                    \
414             ((UINT16)(offsetof(MarshalData_st, TPM2B_PUBLIC_DATA)))
415 #define TPM2B_TEMPLATE_MARSHAL_REF                  \
416             ((UINT16)(offsetof(MarshalData_st, TPM2B_TEMPLATE_DATA)))
417 #define TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF   \
418             ((UINT16)(offsetof(MarshalData_st, TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA)))
419 #define TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF        \
420             ((UINT16)(offsetof(MarshalData_st, TPMU_SENSITIVE_COMPOSITE_DATA)))
421 #define TPMT_SENSITIVE_MARSHAL_REF                  \
422             ((UINT16)(offsetof(MarshalData_st, TPMT_SENSITIVE_DATA)))
423 #define TPM2B_SENSITIVE_MARSHAL_REF                 \
424             ((UINT16)(offsetof(MarshalData_st, TPM2B_SENSITIVE_DATA)))
425 #define TPM2B_PRIVATE_MARSHAL_REF                   \
426             ((UINT16)(offsetof(MarshalData_st, TPM2B_PRIVATE_DATA)))
427 #define TPM2B_ID_OBJECT_MARSHAL_REF                 \
428             ((UINT16)(offsetof(MarshalData_st, TPM2B_ID_OBJECT_DATA)))
429 #define TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF  \
430             ((UINT16)(offsetof(MarshalData_st, TPMS_NV_PIN_COUNTER_PARAMETERS_DATA)))
431 #define TPMA_NV_MARSHAL_REF                         \
432             ((UINT16)(offsetof(MarshalData_st, TPMA_NV_DATA)))
433 #define TPMS_NV_PUBLIC_MARSHAL_REF                  \
434             ((UINT16)(offsetof(MarshalData_st, TPMS_NV_PUBLIC_DATA)))
435 #define TPM2B_NV_PUBLIC_MARSHAL_REF                 \
436             ((UINT16)(offsetof(MarshalData_st, TPM2B_NV_PUBLIC_DATA)))
437 #define TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF         \
438             ((UINT16)(offsetof(MarshalData_st, TPM2B_CONTEXT_SENSITIVE_DATA)))
439 #define TPMS_CONTEXT_DATA_MARSHAL_REF               \
```

```
440             ((UINT16)(offsetof(MarshalData_st, TPMS_CONTEXT_DATA_DATA)))
441 #define TPM2B_CONTEXT_DATA_MARSHAL_REF                  \
442             ((UINT16)(offsetof(MarshalData_st, TPM2B_CONTEXT_DATA_DATA)))
443 #define TPMS_CONTEXT_MARSHAL_REF                        \
444             ((UINT16)(offsetof(MarshalData_st, TPMS_CONTEXT_DATA)))
445 #define TPMS_CREATION_DATA_MARSHAL_REF                  \
446             ((UINT16)(offsetof(MarshalData_st, TPMS_CREATION_DATA_DATA)))
447 #define TPM2B_CREATION_DATA_MARSHAL_REF                 \
448             ((UINT16)(offsetof(MarshalData_st, TPM2B_CREATION_DATA_DATA)))
449 #define TPM_AT_MARSHAL_REF                              \
450             ((UINT16)(offsetof(MarshalData_st, TPM_AT_DATA)))
451 #define TPMS_AC_OUTPUT_MARSHAL_REF                      \
452             ((UINT16)(offsetof(MarshalData_st, TPMS_AC_OUTPUT_DATA)))
453 #define TPML_AC_CAPABILITIES_MARSHAL_REF                \
454             ((UINT16)(offsetof(MarshalData_st, TPML_AC_CAPABILITIES_DATA)))
455 #define Type00_MARSHAL_REF                              \
456             ((UINT16)(offsetof(MarshalData_st, Type00_DATA)))
457 #define Type01_MARSHAL_REF                              \
458             ((UINT16)(offsetof(MarshalData_st, Type01_DATA)))
459 #define Type02_MARSHAL_REF                              \
460             ((UINT16)(offsetof(MarshalData_st, Type02_DATA)))
461 #define Type03_MARSHAL_REF                              \
462             ((UINT16)(offsetof(MarshalData_st, Type03_DATA)))
463 #define Type04_MARSHAL_REF                              \
464             ((UINT16)(offsetof(MarshalData_st, Type04_DATA)))
465 #define Type05_MARSHAL_REF                              \
466             ((UINT16)(offsetof(MarshalData_st, Type05_DATA)))
467 #define Type06_MARSHAL_REF                              \
468             ((UINT16)(offsetof(MarshalData_st, Type06_DATA)))
469 #define Type07_MARSHAL_REF                              \
470             ((UINT16)(offsetof(MarshalData_st, Type07_DATA)))
471 #define Type08_MARSHAL_REF                              \
472             ((UINT16)(offsetof(MarshalData_st, Type08_DATA)))
473 #define Type09_MARSHAL_REF                              Type08_MARSHAL_REF
474 #define Type14_MARSHAL_REF                              Type08_MARSHAL_REF
475 #define Type10_MARSHAL_REF                              \
476             ((UINT16)(offsetof(MarshalData_st, Type10_DATA)))
477 #define Type11_MARSHAL_REF                              \
478             ((UINT16)(offsetof(MarshalData_st, Type11_DATA)))
479 #define Type12_MARSHAL_REF                              \
480             ((UINT16)(offsetof(MarshalData_st, Type12_DATA)))
481 #define Type13_MARSHAL_REF                              \
482             ((UINT16)(offsetof(MarshalData_st, Type13_DATA)))
483 #define Type15_MARSHAL_REF                              \
484             ((UINT16)(offsetof(MarshalData_st, Type15_DATA)))
485 #define Type16_MARSHAL_REF                              Type15_MARSHAL_REF
486 #define Type17_MARSHAL_REF                              \
487             ((UINT16)(offsetof(MarshalData_st, Type17_DATA)))
488 #define Type18_MARSHAL_REF                              \
489             ((UINT16)(offsetof(MarshalData_st, Type18_DATA)))
490 #define Type19_MARSHAL_REF                              \
491             ((UINT16)(offsetof(MarshalData_st, Type19_DATA)))
492 #define Type20_MARSHAL_REF                              \
493             ((UINT16)(offsetof(MarshalData_st, Type20_DATA)))
494 #define Type21_MARSHAL_REF                              Type20_MARSHAL_REF
495 #define Type22_MARSHAL_REF                              \
496             ((UINT16)(offsetof(MarshalData_st, Type22_DATA)))
497 #define Type23_MARSHAL_REF                              \
498             ((UINT16)(offsetof(MarshalData_st, Type23_DATA)))
499 #define Type24_MARSHAL_REF                              \
500             ((UINT16)(offsetof(MarshalData_st, Type24_DATA)))
501 #define Type25_MARSHAL_REF                              \
502             ((UINT16)(offsetof(MarshalData_st, Type25_DATA)))
503 #define Type26_MARSHAL_REF                              \
504             ((UINT16)(offsetof(MarshalData_st, Type26_DATA)))
505 #define Type27_MARSHAL_REF                              \
```

```
506             ((UINT16)(offsetof(MarshalData_st, Type27_DATA)))
507 #define Type28_MARSHAL_REF                          \
508             ((UINT16)(offsetof(MarshalData_st, Type28_DATA)))
509 #define Type29_MARSHAL_REF                          \
510             ((UINT16)(offsetof(MarshalData_st, Type29_DATA)))
511 #define Type30_MARSHAL_REF                          \
512             ((UINT16)(offsetof(MarshalData_st, Type30_DATA)))
513 #define Type31_MARSHAL_REF                          \
514             ((UINT16)(offsetof(MarshalData_st, Type31_DATA)))
515 #define Type32_MARSHAL_REF                          \
516             ((UINT16)(offsetof(MarshalData_st, Type32_DATA)))
517 #define Type33_MARSHAL_REF                          \
518             ((UINT16)(offsetof(MarshalData_st, Type33_DATA)))
519 #define Type34_MARSHAL_REF                          \
520             ((UINT16)(offsetof(MarshalData_st, Type34_DATA)))
521 #define Type35_MARSHAL_REF                          \
522             ((UINT16)(offsetof(MarshalData_st, Type35_DATA)))
523 #define Type36_MARSHAL_REF                          \
524             ((UINT16)(offsetof(MarshalData_st, Type36_DATA)))
525 #define Type37_MARSHAL_REF                          \
526             ((UINT16)(offsetof(MarshalData_st, Type37_DATA)))
527 #define Type38_MARSHAL_REF                          \
528             ((UINT16)(offsetof(MarshalData_st, Type38_DATA)))
529 #define Type39_MARSHAL_REF                          \
530             ((UINT16)(offsetof(MarshalData_st, Type39_DATA)))
531 #define Type40_MARSHAL_REF                          \
532             ((UINT16)(offsetof(MarshalData_st, Type40_DATA)))
533 #define Type41_MARSHAL_REF                          \
534             ((UINT16)(offsetof(MarshalData_st, Type41_DATA)))
535 #define Type42_MARSHAL_REF                          \
536             ((UINT16)(offsetof(MarshalData_st, Type42_DATA)))
537 #define Type43_MARSHAL_REF                          \
538             ((UINT16)(offsetof(MarshalData_st, Type43_DATA)))
539 #define Type44_MARSHAL_REF                          \
540             ((UINT16)(offsetof(MarshalData_st, Type44_DATA)))
541 //#defines to change calling sequence for code using marshaling
542 #define UINT8_Unmarshal(target, buffer, size)                       \
543     Unmarshal(UINT8_MARSHAL_REF, (target), (buffer), (size))
544 #define UINT8_Marshal(source, buffer, size)                         \
545     Marshal(UINT8_MARSHAL_REF, (source), (buffer), (size))
546 #define BYTE_Unmarshal(target, buffer, size)                        \
547     Unmarshal(UINT8_MARSHAL_REF, (target), (buffer), (size))
548 #define BYTE_Marshal(source, buffer, size)                          \
549     Marshal(UINT8_MARSHAL_REF, (source), (buffer), (size))
550 #define INT8_Unmarshal(target, buffer, size)                        \
551     Unmarshal(INT8_MARSHAL_REF, (target), (buffer), (size))
552 #define INT8_Marshal(source, buffer, size)                          \
553     Marshal(INT8_MARSHAL_REF, (source), (buffer), (size))
554 #define UINT16_Unmarshal(target, buffer, size)                      \
555     Unmarshal(UINT16_MARSHAL_REF, (target), (buffer), (size))
556 #define UINT16_Marshal(source, buffer, size)                        \
557     Marshal(UINT16_MARSHAL_REF, (source), (buffer), (size))
558 #define INT16_Unmarshal(target, buffer, size)                       \
559     Unmarshal(INT16_MARSHAL_REF, (target), (buffer), (size))
560 #define INT16_Marshal(source, buffer, size)                         \
561     Marshal(INT16_MARSHAL_REF, (source), (buffer), (size))
562 #define UINT32_Unmarshal(target, buffer, size)                      \
563     Unmarshal(UINT32_MARSHAL_REF, (target), (buffer), (size))
564 #define UINT32_Marshal(source, buffer, size)                        \
565     Marshal(UINT32_MARSHAL_REF, (source), (buffer), (size))
566 #define INT32_Unmarshal(target, buffer, size)                       \
567     Unmarshal(INT32_MARSHAL_REF, (target), (buffer), (size))
568 #define INT32_Marshal(source, buffer, size)                         \
569     Marshal(INT32_MARSHAL_REF, (source), (buffer), (size))
570 #define UINT64_Unmarshal(target, buffer, size)                      \
571     Unmarshal(UINT64_MARSHAL_REF, (target), (buffer), (size))
```

```
572 #define UINT64_Marshal(source, buffer, size)                                    \
573     Marshal(UINT64_MARSHAL_REF, (source), (buffer), (size))
574 #define INT64_Unmarshal(target, buffer, size)                                   \
575     Unmarshal(INT64_MARSHAL_REF, (target), (buffer), (size))
576 #define INT64_Marshal(source, buffer, size)                                     \
577     Marshal(INT64_MARSHAL_REF, (source), (buffer), (size))
578 #define TPM_ALGORITHM_ID_Unmarshal(target, buffer, size)                        \
579     Unmarshal(TPM_ALGORITHM_ID_MARSHAL_REF, (target), (buffer), (size))
580 #define TPM_ALGORITHM_ID_Marshal(source, buffer, size)                          \
581     Marshal(TPM_ALGORITHM_ID_MARSHAL_REF, (source), (buffer), (size))
582 #define TPM_MODIFIER_INDICATOR_Unmarshal(target, buffer, size)                  \
583     Unmarshal(TPM_MODIFIER_INDICATOR_MARSHAL_REF, (target), (buffer), (size))
584 #define TPM_MODIFIER_INDICATOR_Marshal(source, buffer, size)                    \
585     Marshal(TPM_MODIFIER_INDICATOR_MARSHAL_REF, (source), (buffer), (size))
586 #define TPM_AUTHORIZATION_SIZE_Unmarshal(target, buffer, size)                  \
587     Unmarshal(TPM_AUTHORIZATION_SIZE_MARSHAL_REF, (target), (buffer), (size))
588 #define TPM_AUTHORIZATION_SIZE_Marshal(source, buffer, size)                    \
589     Marshal(TPM_AUTHORIZATION_SIZE_MARSHAL_REF, (source), (buffer), (size))
590 #define TPM_PARAMETER_SIZE_Unmarshal(target, buffer, size)                      \
591     Unmarshal(TPM_PARAMETER_SIZE_MARSHAL_REF, (target), (buffer), (size))
592 #define TPM_PARAMETER_SIZE_Marshal(source, buffer, size)                        \
593     Marshal(TPM_PARAMETER_SIZE_MARSHAL_REF, (source), (buffer), (size))
594 #define TPM_KEY_SIZE_Unmarshal(target, buffer, size)                            \
595     Unmarshal(TPM_KEY_SIZE_MARSHAL_REF, (target), (buffer), (size))
596 #define TPM_KEY_SIZE_Marshal(source, buffer, size)                              \
597     Marshal(TPM_KEY_SIZE_MARSHAL_REF, (source), (buffer), (size))
598 #define TPM_KEY_BITS_Unmarshal(target, buffer, size)                            \
599     Unmarshal(TPM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
600 #define TPM_KEY_BITS_Marshal(source, buffer, size)                              \
601     Marshal(TPM_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
602 #define TPM_GENERATED_Marshal(source, buffer, size)                             \
603     Marshal(TPM_GENERATED_MARSHAL_REF, (source), (buffer), (size))
604 #define TPM_ALG_ID_Unmarshal(target, buffer, size)                              \
605     Unmarshal(TPM_ALG_ID_MARSHAL_REF, (target), (buffer), (size))
606 #define TPM_ALG_ID_Marshal(source, buffer, size)                                \
607     Marshal(TPM_ALG_ID_MARSHAL_REF, (source), (buffer), (size))
608 #define TPM_ECC_CURVE_Unmarshal(target, buffer, size)                           \
609     Unmarshal(TPM_ECC_CURVE_MARSHAL_REF, (target), (buffer), (size))
610 #define TPM_ECC_CURVE_Marshal(source, buffer, size)                             \
611     Marshal(TPM_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
612 #define TPM_CC_Unmarshal(target, buffer, size)                                  \
613     Unmarshal(TPM_CC_MARSHAL_REF, (target), (buffer), (size))
614 #define TPM_CC_Marshal(source, buffer, size)                                    \
615     Marshal(TPM_CC_MARSHAL_REF, (source), (buffer), (size))
616 #define TPM_RC_Marshal(source, buffer, size)                                    \
617     Marshal(TPM_RC_MARSHAL_REF, (source), (buffer), (size))
618 #define TPM_CLOCK_ADJUST_Unmarshal(target, buffer, size)                        \
619     Unmarshal(TPM_CLOCK_ADJUST_MARSHAL_REF, (target), (buffer), (size))
620 #define TPM_EO_Unmarshal(target, buffer, size)                                  \
621     Unmarshal(TPM_EO_MARSHAL_REF, (target), (buffer), (size))
622 #define TPM_EO_Marshal(source, buffer, size)                                    \
623     Marshal(TPM_EO_MARSHAL_REF, (source), (buffer), (size))
624 #define TPM_ST_Unmarshal(target, buffer, size)                                  \
625     Unmarshal(TPM_ST_MARSHAL_REF, (target), (buffer), (size))
626 #define TPM_ST_Marshal(source, buffer, size)                                    \
627     Marshal(TPM_ST_MARSHAL_REF, (source), (buffer), (size))
628 #define TPM_SU_Unmarshal(target, buffer, size)                                  \
629     Unmarshal(TPM_SU_MARSHAL_REF, (target), (buffer), (size))
630 #define TPM_SE_Unmarshal(target, buffer, size)                                  \
631     Unmarshal(TPM_SE_MARSHAL_REF, (target), (buffer), (size))
632 #define TPM_CAP_Unmarshal(target, buffer, size)                                 \
633     Unmarshal(TPM_CAP_MARSHAL_REF, (target), (buffer), (size))
634 #define TPM_CAP_Marshal(source, buffer, size)                                   \
635     Marshal(TPM_CAP_MARSHAL_REF, (source), (buffer), (size))
636 #define TPM_PT_Unmarshal(target, buffer, size)                                  \
637     Unmarshal(TPM_PT_MARSHAL_REF, (target), (buffer), (size))
```

```
638  #define TPM_PT_Marshal(source, buffer, size)                                    \
639      Marshal(TPM_PT_MARSHAL_REF, (source), (buffer), (size))
640  #define TPM_PT_PCR_Unmarshal(target, buffer, size)                              \
641      Unmarshal(TPM_PT_PCR_MARSHAL_REF, (target), (buffer), (size))
642  #define TPM_PT_PCR_Marshal(source, buffer, size)                                \
643      Marshal(TPM_PT_PCR_MARSHAL_REF, (source), (buffer), (size))
644  #define TPM_PS_Marshal(source, buffer, size)                                    \
645      Marshal(TPM_PS_MARSHAL_REF, (source), (buffer), (size))
646  #define TPM_HANDLE_Unmarshal(target, buffer, size)                              \
647      Unmarshal(TPM_HANDLE_MARSHAL_REF, (target), (buffer), (size))
648  #define TPM_HANDLE_Marshal(source, buffer, size)                                \
649      Marshal(TPM_HANDLE_MARSHAL_REF, (source), (buffer), (size))
650  #define TPM_HT_Unmarshal(target, buffer, size)                                  \
651      Unmarshal(TPM_HT_MARSHAL_REF, (target), (buffer), (size))
652  #define TPM_HT_Marshal(source, buffer, size)                                    \
653      Marshal(TPM_HT_MARSHAL_REF, (source), (buffer), (size))
654  #define TPM_RH_Unmarshal(target, buffer, size)                                  \
655      Unmarshal(TPM_RH_MARSHAL_REF, (target), (buffer), (size))
656  #define TPM_RH_Marshal(source, buffer, size)                                    \
657      Marshal(TPM_RH_MARSHAL_REF, (source), (buffer), (size))
658  #define TPM_HC_Unmarshal(target, buffer, size)                                  \
659      Unmarshal(TPM_HC_MARSHAL_REF, (target), (buffer), (size))
660  #define TPM_HC_Marshal(source, buffer, size)                                    \
661      Marshal(TPM_HC_MARSHAL_REF, (source), (buffer), (size))
662  #define TPMA_ALGORITHM_Unmarshal(target, buffer, size)                          \
663      Unmarshal(TPMA_ALGORITHM_MARSHAL_REF, (target), (buffer), (size))
664  #define TPMA_ALGORITHM_Marshal(source, buffer, size)                            \
665      Marshal(TPMA_ALGORITHM_MARSHAL_REF, (source), (buffer), (size))
666  #define TPMA_OBJECT_Unmarshal(target, buffer, size)                             \
667      Unmarshal(TPMA_OBJECT_MARSHAL_REF, (target), (buffer), (size))
668  #define TPMA_OBJECT_Marshal(source, buffer, size)                               \
669      Marshal(TPMA_OBJECT_MARSHAL_REF, (source), (buffer), (size))
670  #define TPMA_SESSION_Unmarshal(target, buffer, size)                            \
671      Unmarshal(TPMA_SESSION_MARSHAL_REF, (target), (buffer), (size))
672  #define TPMA_SESSION_Marshal(source, buffer, size)                              \
673      Marshal(TPMA_SESSION_MARSHAL_REF, (source), (buffer), (size))
674  #define TPMA_LOCALITY_Unmarshal(target, buffer, size)                           \
675      Unmarshal(TPMA_LOCALITY_MARSHAL_REF, (target), (buffer), (size))
676  #define TPMA_LOCALITY_Marshal(source, buffer, size)                             \
677      Marshal(TPMA_LOCALITY_MARSHAL_REF, (source), (buffer), (size))
678  #define TPMA_PERMANENT_Marshal(source, buffer, size)                            \
679      Marshal(TPMA_PERMANENT_MARSHAL_REF, (source), (buffer), (size))
680  #define TPMA_STARTUP_CLEAR_Marshal(source, buffer, size)                        \
681      Marshal(TPMA_STARTUP_CLEAR_MARSHAL_REF, (source), (buffer), (size))
682  #define TPMA_MEMORY_Marshal(source, buffer, size)                               \
683      Marshal(TPMA_MEMORY_MARSHAL_REF, (source), (buffer), (size))
684  #define TPMA_CC_Marshal(source, buffer, size)                                   \
685      Marshal(TPMA_CC_MARSHAL_REF, (source), (buffer), (size))
686  #define TPMA_MODES_Marshal(source, buffer, size)                                \
687      Marshal(TPMA_MODES_MARSHAL_REF, (source), (buffer), (size))
688  #define TPMA_X509_KEY_USAGE_Marshal(source, buffer, size)                       \
689      Marshal(TPMA_X509_KEY_USAGE_MARSHAL_REF, (source), (buffer), (size))
690  #define TPMA_ACT_Unmarshal(target, buffer, size)                                \
691      Unmarshal(TPMA_ACT_MARSHAL_REF, (target), (buffer), (size))
692  #define TPMA_ACT_Marshal(source, buffer, size)                                  \
693      Marshal(TPMA_ACT_MARSHAL_REF, (source), (buffer), (size))
694  #define TPMI_YES_NO_Unmarshal(target, buffer, size)                             \
695      Unmarshal(TPMI_YES_NO_MARSHAL_REF, (target), (buffer), (size))
696  #define TPMI_YES_NO_Marshal(source, buffer, size)                               \
697      Marshal(TPMI_YES_NO_MARSHAL_REF, (source), (buffer), (size))
698  #define TPMI_DH_OBJECT_Unmarshal(target, buffer, size, flag)                    \
699      Unmarshal(TPMI_DH_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
700        (size))
701  #define TPMI_DH_OBJECT_Marshal(source, buffer, size)                            \
702      Marshal(TPMI_DH_OBJECT_MARSHAL_REF, (source), (buffer), (size))
703  #define TPMI_DH_PARENT_Unmarshal(target, buffer, size, flag)                    \
```

```
704         Unmarshal(TPMI_DH_PARENT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
705           (size))
706     #define TPMI_DH_PARENT_Marshal(source, buffer, size)                                    \
707         Marshal(TPMI_DH_PARENT_MARSHAL_REF, (source), (buffer), (size))
708     #define TPMI_DH_PERSISTENT_Unmarshal(target, buffer, size)                              \
709         Unmarshal(TPMI_DH_PERSISTENT_MARSHAL_REF, (target), (buffer), (size))
710     #define TPMI_DH_PERSISTENT_Marshal(source, buffer, size)                                \
711         Marshal(TPMI_DH_PERSISTENT_MARSHAL_REF, (source), (buffer), (size))
712     #define TPMI_DH_ENTITY_Unmarshal(target, buffer, size, flag)                            \
713         Unmarshal(TPMI_DH_ENTITY_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
714           (size))
715     #define TPMI_DH_PCR_Unmarshal(target, buffer, size, flag)                               \
716         Unmarshal(TPMI_DH_PCR_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),   \
717           (size))
718     #define TPMI_SH_AUTH_SESSION_Unmarshal(target, buffer, size, flag)                      \
719         Unmarshal(TPMI_SH_AUTH_SESSION_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
720           (buffer), (size))
721     #define TPMI_SH_AUTH_SESSION_Marshal(source, buffer, size)                              \
722         Marshal(TPMI_SH_AUTH_SESSION_MARSHAL_REF, (source), (buffer), (size))
723     #define TPMI_SH_HMAC_Unmarshal(target, buffer, size)                                    \
724         Unmarshal(TPMI_SH_HMAC_MARSHAL_REF, (target), (buffer), (size))
725     #define TPMI_SH_HMAC_Marshal(source, buffer, size)                                      \
726         Marshal(TPMI_SH_HMAC_MARSHAL_REF, (source), (buffer), (size))
727     #define TPMI_SH_POLICY_Unmarshal(target, buffer, size)                                  \
728         Unmarshal(TPMI_SH_POLICY_MARSHAL_REF, (target), (buffer), (size))
729     #define TPMI_SH_POLICY_Marshal(source, buffer, size)                                    \
730         Marshal(TPMI_SH_POLICY_MARSHAL_REF, (source), (buffer), (size))
731     #define TPMI_DH_CONTEXT_Unmarshal(target, buffer, size)                                 \
732         Unmarshal(TPMI_DH_CONTEXT_MARSHAL_REF, (target), (buffer), (size))
733     #define TPMI_DH_CONTEXT_Marshal(source, buffer, size)                                   \
734         Marshal(TPMI_DH_CONTEXT_MARSHAL_REF, (source), (buffer), (size))
735     #define TPMI_DH_SAVED_Unmarshal(target, buffer, size)                                   \
736         Unmarshal(TPMI_DH_SAVED_MARSHAL_REF, (target), (buffer), (size))
737     #define TPMI_DH_SAVED_Marshal(source, buffer, size)                                     \
738         Marshal(TPMI_DH_SAVED_MARSHAL_REF, (source), (buffer), (size))
739     #define TPMI_RH_HIERARCHY_Unmarshal(target, buffer, size, flag)                         \
740         Unmarshal(TPMI_RH_HIERARCHY_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),       \
741           (buffer), (size))
742     #define TPMI_RH_HIERARCHY_Marshal(source, buffer, size)                                 \
743         Marshal(TPMI_RH_HIERARCHY_MARSHAL_REF, (source), (buffer), (size))
744     #define TPMI_RH_ENABLES_Unmarshal(target, buffer, size, flag)                           \
745         Unmarshal(TPMI_RH_ENABLES_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),         \
746           (buffer), (size))
747     #define TPMI_RH_ENABLES_Marshal(source, buffer, size)                                   \
748         Marshal(TPMI_RH_ENABLES_MARSHAL_REF, (source), (buffer), (size))
749     #define TPMI_RH_HIERARCHY_AUTH_Unmarshal(target, buffer, size)                          \
750         Unmarshal(TPMI_RH_HIERARCHY_AUTH_MARSHAL_REF, (target), (buffer), (size))
751     #define TPMI_RH_HIERARCHY_POLICY_Unmarshal(target, buffer, size)                        \
752         Unmarshal(TPMI_RH_HIERARCHY_POLICY_MARSHAL_REF, (target), (buffer), (size))
753     #define TPMI_RH_PLATFORM_Unmarshal(target, buffer, size)                                \
754         Unmarshal(TPMI_RH_PLATFORM_MARSHAL_REF, (target), (buffer), (size))
755     #define TPMI_RH_OWNER_Unmarshal(target, buffer, size, flag)                             \
756         Unmarshal(TPMI_RH_OWNER_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
757           (size))
758     #define TPMI_RH_ENDORSEMENT_Unmarshal(target, buffer, size, flag)                       \
759         Unmarshal(TPMI_RH_ENDORSEMENT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
760           (buffer), (size))
761     #define TPMI_RH_PROVISION_Unmarshal(target, buffer, size)                               \
762         Unmarshal(TPMI_RH_PROVISION_MARSHAL_REF, (target), (buffer), (size))
763     #define TPMI_RH_CLEAR_Unmarshal(target, buffer, size)                                   \
764         Unmarshal(TPMI_RH_CLEAR_MARSHAL_REF, (target), (buffer), (size))
765     #define TPMI_RH_NV_AUTH_Unmarshal(target, buffer, size)                                 \
766         Unmarshal(TPMI_RH_NV_AUTH_MARSHAL_REF, (target), (buffer), (size))
767     #define TPMI_RH_LOCKOUT_Unmarshal(target, buffer, size)                                 \
768         Unmarshal(TPMI_RH_LOCKOUT_MARSHAL_REF, (target), (buffer), (size))
769     #define TPMI_RH_NV_INDEX_Unmarshal(target, buffer, size)                                \
```

```
770     Unmarshal(TPMI_RH_NV_INDEX_MARSHAL_REF, (target), (buffer), (size))
771 #define TPMI_RH_NV_INDEX_Marshal(source, buffer, size)                          \
772     Marshal(TPMI_RH_NV_INDEX_MARSHAL_REF, (source), (buffer), (size))
773 #define TPMI_RH_AC_Unmarshal(target, buffer, size)                              \
774     Unmarshal(TPMI_RH_AC_MARSHAL_REF, (target), (buffer), (size))
775 #define TPMI_RH_ACT_Unmarshal(target, buffer, size)                             \
776     Unmarshal(TPMI_RH_ACT_MARSHAL_REF, (target), (buffer), (size))
777 #define TPMI_RH_ACT_Marshal(source, buffer, size)                               \
778     Marshal(TPMI_RH_ACT_MARSHAL_REF, (source), (buffer), (size))
779 #define TPMI_ALG_HASH_Unmarshal(target, buffer, size, flag)                     \
780     Unmarshal(TPMI_ALG_HASH_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
781       (size))
782 #define TPMI_ALG_HASH_Marshal(source, buffer, size)                             \
783     Marshal(TPMI_ALG_HASH_MARSHAL_REF, (source), (buffer), (size))
784 #define TPMI_ALG_ASYM_Unmarshal(target, buffer, size, flag)                     \
785     Unmarshal(TPMI_ALG_ASYM_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
786       (size))
787 #define TPMI_ALG_ASYM_Marshal(source, buffer, size)                             \
788     Marshal(TPMI_ALG_ASYM_MARSHAL_REF, (source), (buffer), (size))
789 #define TPMI_ALG_SYM_Unmarshal(target, buffer, size, flag)                      \
790     Unmarshal(TPMI_ALG_SYM_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),  \
791       (size))
792 #define TPMI_ALG_SYM_Marshal(source, buffer, size)                              \
793     Marshal(TPMI_ALG_SYM_MARSHAL_REF, (source), (buffer), (size))
794 #define TPMI_ALG_SYM_OBJECT_Unmarshal(target, buffer, size, flag)               \
795     Unmarshal(TPMI_ALG_SYM_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
796       (buffer), (size))
797 #define TPMI_ALG_SYM_OBJECT_Marshal(source, buffer, size)                       \
798     Marshal(TPMI_ALG_SYM_OBJECT_MARSHAL_REF, (source), (buffer), (size))
799 #define TPMI_ALG_SYM_MODE_Unmarshal(target, buffer, size, flag)                 \
800     Unmarshal(TPMI_ALG_SYM_MODE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),       \
801       (buffer), (size))
802 #define TPMI_ALG_SYM_MODE_Marshal(source, buffer, size)                         \
803     Marshal(TPMI_ALG_SYM_MODE_MARSHAL_REF, (source), (buffer), (size))
804 #define TPMI_ALG_KDF_Unmarshal(target, buffer, size, flag)                      \
805     Unmarshal(TPMI_ALG_KDF_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),  \
806       (size))
807 #define TPMI_ALG_KDF_Marshal(source, buffer, size)                              \
808     Marshal(TPMI_ALG_KDF_MARSHAL_REF, (source), (buffer), (size))
809 #define TPMI_ALG_SIG_SCHEME_Unmarshal(target, buffer, size, flag)               \
810     Unmarshal(TPMI_ALG_SIG_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
811       (buffer), (size))
812 #define TPMI_ALG_SIG_SCHEME_Marshal(source, buffer, size)                       \
813     Marshal(TPMI_ALG_SIG_SCHEME_MARSHAL_REF, (source), (buffer), (size))
814 #define TPMI_ECC_KEY_EXCHANGE_Unmarshal(target, buffer, size, flag)             \
815     Unmarshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),   \
816       (buffer), (size))
817 #define TPMI_ECC_KEY_EXCHANGE_Marshal(source, buffer, size)                     \
818     Marshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF, (source), (buffer), (size))
819 #define TPMI_ST_COMMAND_TAG_Unmarshal(target, buffer, size)                     \
820     Unmarshal(TPMI_ST_COMMAND_TAG_MARSHAL_REF, (target), (buffer), (size))
821 #define TPMI_ST_COMMAND_TAG_Marshal(source, buffer, size)                       \
822     Marshal(TPMI_ST_COMMAND_TAG_MARSHAL_REF, (source), (buffer), (size))
823 #define TPMI_ALG_MAC_SCHEME_Unmarshal(target, buffer, size, flag)               \
824     Unmarshal(TPMI_ALG_MAC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
825       (buffer), (size))
826 #define TPMI_ALG_MAC_SCHEME_Marshal(source, buffer, size)                       \
827     Marshal(TPMI_ALG_MAC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
828 #define TPMI_ALG_CIPHER_MODE_Unmarshal(target, buffer, size, flag)              \
829     Unmarshal(TPMI_ALG_CIPHER_MODE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
830       (buffer), (size))
831 #define TPMI_ALG_CIPHER_MODE_Marshal(source, buffer, size)                      \
832     Marshal(TPMI_ALG_CIPHER_MODE_MARSHAL_REF, (source), (buffer), (size))
833 #define TPMS_EMPTY_Unmarshal(target, buffer, size)                              \
834     Unmarshal(TPMS_EMPTY_MARSHAL_REF, (target), (buffer), (size))
835 #define TPMS_EMPTY_Marshal(source, buffer, size)                                \
```

```
836         Marshal(TPMS_EMPTY_MARSHAL_REF, (source), (buffer), (size))
837     #define TPMS_ALGORITHM_DESCRIPTION_Marshal(source, buffer, size)                    \
838         Marshal(TPMS_ALGORITHM_DESCRIPTION_MARSHAL_REF, (source), (buffer), (size))
839     #define TPMU_HA_Unmarshal(target, buffer, size, selector)                           \
840         UnmarshalUnion(TPMU_HA_MARSHAL_REF, (target), (buffer), (size), (selector))
841     #define TPMU_HA_Marshal(source, buffer, size, selector)                             \
842         MarshalUnion(TPMU_HA_MARSHAL_REF, (target), (buffer), (size), (selector))
843     #define TPMT_HA_Unmarshal(target, buffer, size, flag)                               \
844         Unmarshal(TPMT_HA_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),       \
845           (size))
846     #define TPMT_HA_Marshal(source, buffer, size)                                       \
847         Marshal(TPMT_HA_MARSHAL_REF, (source), (buffer), (size))
848     #define TPM2B_DIGEST_Unmarshal(target, buffer, size)                                \
849         Unmarshal(TPM2B_DIGEST_MARSHAL_REF, (target), (buffer), (size))
850     #define TPM2B_DIGEST_Marshal(source, buffer, size)                                  \
851         Marshal(TPM2B_DIGEST_MARSHAL_REF, (source), (buffer), (size))
852     #define TPM2B_DATA_Unmarshal(target, buffer, size)                                  \
853         Unmarshal(TPM2B_DATA_MARSHAL_REF, (target), (buffer), (size))
854     #define TPM2B_DATA_Marshal(source, buffer, size)                                    \
855         Marshal(TPM2B_DATA_MARSHAL_REF, (source), (buffer), (size))
856     #define TPM2B_NONCE_Unmarshal(target, buffer, size)                                 \
857         Unmarshal(TPM2B_NONCE_MARSHAL_REF, (target), (buffer), (size))
858     #define TPM2B_NONCE_Marshal(source, buffer, size)                                   \
859         Marshal(TPM2B_NONCE_MARSHAL_REF, (source), (buffer), (size))
860     #define TPM2B_AUTH_Unmarshal(target, buffer, size)                                  \
861         Unmarshal(TPM2B_AUTH_MARSHAL_REF, (target), (buffer), (size))
862     #define TPM2B_AUTH_Marshal(source, buffer, size)                                    \
863         Marshal(TPM2B_AUTH_MARSHAL_REF, (source), (buffer), (size))
864     #define TPM2B_OPERAND_Unmarshal(target, buffer, size)                               \
865         Unmarshal(TPM2B_OPERAND_MARSHAL_REF, (target), (buffer), (size))
866     #define TPM2B_OPERAND_Marshal(source, buffer, size)                                 \
867         Marshal(TPM2B_OPERAND_MARSHAL_REF, (source), (buffer), (size))
868     #define TPM2B_EVENT_Unmarshal(target, buffer, size)                                 \
869         Unmarshal(TPM2B_EVENT_MARSHAL_REF, (target), (buffer), (size))
870     #define TPM2B_EVENT_Marshal(source, buffer, size)                                   \
871         Marshal(TPM2B_EVENT_MARSHAL_REF, (source), (buffer), (size))
872     #define TPM2B_MAX_BUFFER_Unmarshal(target, buffer, size)                            \
873         Unmarshal(TPM2B_MAX_BUFFER_MARSHAL_REF, (target), (buffer), (size))
874     #define TPM2B_MAX_BUFFER_Marshal(source, buffer, size)                              \
875         Marshal(TPM2B_MAX_BUFFER_MARSHAL_REF, (source), (buffer), (size))
876     #define TPM2B_MAX_NV_BUFFER_Unmarshal(target, buffer, size)                         \
877         Unmarshal(TPM2B_MAX_NV_BUFFER_MARSHAL_REF, (target), (buffer), (size))
878     #define TPM2B_MAX_NV_BUFFER_Marshal(source, buffer, size)                           \
879         Marshal(TPM2B_MAX_NV_BUFFER_MARSHAL_REF, (source), (buffer), (size))
880     #define TPM2B_TIMEOUT_Unmarshal(target, buffer, size)                               \
881         Unmarshal(TPM2B_TIMEOUT_MARSHAL_REF, (target), (buffer), (size))
882     #define TPM2B_TIMEOUT_Marshal(source, buffer, size)                                 \
883         Marshal(TPM2B_TIMEOUT_MARSHAL_REF, (source), (buffer), (size))
884     #define TPM2B_IV_Unmarshal(target, buffer, size)                                    \
885         Unmarshal(TPM2B_IV_MARSHAL_REF, (target), (buffer), (size))
886     #define TPM2B_IV_Marshal(source, buffer, size)                                      \
887         Marshal(TPM2B_IV_MARSHAL_REF, (source), (buffer), (size))
888     #define TPM2B_NAME_Unmarshal(target, buffer, size)                                  \
889         Unmarshal(TPM2B_NAME_MARSHAL_REF, (target), (buffer), (size))
890     #define TPM2B_NAME_Marshal(source, buffer, size)                                    \
891         Marshal(TPM2B_NAME_MARSHAL_REF, (source), (buffer), (size))
892     #define TPMS_PCR_SELECT_Unmarshal(target, buffer, size)                             \
893         Unmarshal(TPMS_PCR_SELECT_MARSHAL_REF, (target), (buffer), (size))
894     #define TPMS_PCR_SELECT_Marshal(source, buffer, size)                               \
895         Marshal(TPMS_PCR_SELECT_MARSHAL_REF, (source), (buffer), (size))
896     #define TPMS_PCR_SELECTION_Unmarshal(target, buffer, size)                          \
897         Unmarshal(TPMS_PCR_SELECTION_MARSHAL_REF, (target), (buffer), (size))
898     #define TPMS_PCR_SELECTION_Marshal(source, buffer, size)                            \
899         Marshal(TPMS_PCR_SELECTION_MARSHAL_REF, (source), (buffer), (size))
900     #define TPMT_TK_CREATION_Unmarshal(target, buffer, size)                            \
901         Unmarshal(TPMT_TK_CREATION_MARSHAL_REF, (target), (buffer), (size))
```

```
902  #define TPMT_TK_CREATION_Marshal(source, buffer, size)                          \
903      Marshal(TPMT_TK_CREATION_MARSHAL_REF, (source), (buffer), (size))
904  #define TPMT_TK_VERIFIED_Unmarshal(target, buffer, size)                        \
905      Unmarshal(TPMT_TK_VERIFIED_MARSHAL_REF, (target), (buffer), (size))
906  #define TPMT_TK_VERIFIED_Marshal(source, buffer, size)                          \
907      Marshal(TPMT_TK_VERIFIED_MARSHAL_REF, (source), (buffer), (size))
908  #define TPMT_TK_AUTH_Unmarshal(target, buffer, size)                            \
909      Unmarshal(TPMT_TK_AUTH_MARSHAL_REF, (target), (buffer), (size))
910  #define TPMT_TK_AUTH_Marshal(source, buffer, size)                              \
911      Marshal(TPMT_TK_AUTH_MARSHAL_REF, (source), (buffer), (size))
912  #define TPMT_TK_HASHCHECK_Unmarshal(target, buffer, size)                       \
913      Unmarshal(TPMT_TK_HASHCHECK_MARSHAL_REF, (target), (buffer), (size))
914  #define TPMT_TK_HASHCHECK_Marshal(source, buffer, size)                         \
915      Marshal(TPMT_TK_HASHCHECK_MARSHAL_REF, (source), (buffer), (size))
916  #define TPMS_ALG_PROPERTY_Marshal(source, buffer, size)                         \
917      Marshal(TPMS_ALG_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
918  #define TPMS_TAGGED_PROPERTY_Marshal(source, buffer, size)                      \
919      Marshal(TPMS_TAGGED_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
920  #define TPMS_TAGGED_PCR_SELECT_Marshal(source, buffer, size)                    \
921      Marshal(TPMS_TAGGED_PCR_SELECT_MARSHAL_REF, (source), (buffer), (size))
922  #define TPMS_TAGGED_POLICY_Marshal(source, buffer, size)                        \
923      Marshal(TPMS_TAGGED_POLICY_MARSHAL_REF, (source), (buffer), (size))
924  #define TPMS_ACT_DATA_Marshal(source, buffer, size)                             \
925      Marshal(TPMS_ACT_DATA_MARSHAL_REF, (source), (buffer), (size))
926  #define TPML_CC_Unmarshal(target, buffer, size)                                 \
927      Unmarshal(TPML_CC_MARSHAL_REF, (target), (buffer), (size))
928  #define TPML_CC_Marshal(source, buffer, size)                                   \
929      Marshal(TPML_CC_MARSHAL_REF, (source), (buffer), (size))
930  #define TPML_CCA_Marshal(source, buffer, size)                                  \
931      Marshal(TPML_CCA_MARSHAL_REF, (source), (buffer), (size))
932  #define TPML_ALG_Unmarshal(target, buffer, size)                                \
933      Unmarshal(TPML_ALG_MARSHAL_REF, (target), (buffer), (size))
934  #define TPML_ALG_Marshal(source, buffer, size)                                  \
935      Marshal(TPML_ALG_MARSHAL_REF, (source), (buffer), (size))
936  #define TPML_HANDLE_Marshal(source, buffer, size)                               \
937      Marshal(TPML_HANDLE_MARSHAL_REF, (source), (buffer), (size))
938  #define TPML_DIGEST_Unmarshal(target, buffer, size)                             \
939      Unmarshal(TPML_DIGEST_MARSHAL_REF, (target), (buffer), (size))
940  #define TPML_DIGEST_Marshal(source, buffer, size)                               \
941      Marshal(TPML_DIGEST_MARSHAL_REF, (source), (buffer), (size))
942  #define TPML_DIGEST_VALUES_Unmarshal(target, buffer, size)                      \
943      Unmarshal(TPML_DIGEST_VALUES_MARSHAL_REF, (target), (buffer), (size))
944  #define TPML_DIGEST_VALUES_Marshal(source, buffer, size)                        \
945      Marshal(TPML_DIGEST_VALUES_MARSHAL_REF, (source), (buffer), (size))
946  #define TPML_PCR_SELECTION_Unmarshal(target, buffer, size)                      \
947      Unmarshal(TPML_PCR_SELECTION_MARSHAL_REF, (target), (buffer), (size))
948  #define TPML_PCR_SELECTION_Marshal(source, buffer, size)                        \
949      Marshal(TPML_PCR_SELECTION_MARSHAL_REF, (source), (buffer), (size))
950  #define TPML_ALG_PROPERTY_Marshal(source, buffer, size)                         \
951      Marshal(TPML_ALG_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
952  #define TPML_TAGGED_TPM_PROPERTY_Marshal(source, buffer, size)                  \
953      Marshal(TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
954  #define TPML_TAGGED_PCR_PROPERTY_Marshal(source, buffer, size)                  \
955      Marshal(TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
956  #define TPML_ECC_CURVE_Marshal(source, buffer, size)                            \
957      Marshal(TPML_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
958  #define TPML_TAGGED_POLICY_Marshal(source, buffer, size)                        \
959      Marshal(TPML_TAGGED_POLICY_MARSHAL_REF, (source), (buffer), (size))
960  #define TPML_ACT_DATA_Marshal(source, buffer, size)                             \
961      Marshal(TPML_ACT_DATA_MARSHAL_REF, (source), (buffer), (size))
962  #define TPMU_CAPABILITIES_Marshal(source, buffer, size, selector)               \
963      MarshalUnion(TPMU_CAPABILITIES_MARSHAL_REF, (target), (buffer), (size),     \
964        (selector))
965  #define TPMS_CAPABILITY_DATA_Marshal(source, buffer, size)                      \
966      Marshal(TPMS_CAPABILITY_DATA_MARSHAL_REF, (source), (buffer), (size))
967  #define TPMS_CLOCK_INFO_Unmarshal(target, buffer, size)                         \
```

```
968         Unmarshal(TPMS_CLOCK_INFO_MARSHAL_REF, (target), (buffer), (size))
969     #define TPMS_CLOCK_INFO_Marshal(source, buffer, size)                                   \
970         Marshal(TPMS_CLOCK_INFO_MARSHAL_REF, (source), (buffer), (size))
971     #define TPMS_TIME_INFO_Unmarshal(target, buffer, size)                                  \
972         Unmarshal(TPMS_TIME_INFO_MARSHAL_REF, (target), (buffer), (size))
973     #define TPMS_TIME_INFO_Marshal(source, buffer, size)                                    \
974         Marshal(TPMS_TIME_INFO_MARSHAL_REF, (source), (buffer), (size))
975     #define TPMS_TIME_ATTEST_INFO_Marshal(source, buffer, size)                             \
976         Marshal(TPMS_TIME_ATTEST_INFO_MARSHAL_REF, (source), (buffer), (size))
977     #define TPMS_CERTIFY_INFO_Marshal(source, buffer, size)                                 \
978         Marshal(TPMS_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
979     #define TPMS_QUOTE_INFO_Marshal(source, buffer, size)                                   \
980         Marshal(TPMS_QUOTE_INFO_MARSHAL_REF, (source), (buffer), (size))
981     #define TPMS_COMMAND_AUDIT_INFO_Marshal(source, buffer, size)                           \
982         Marshal(TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF, (source), (buffer), (size))
983     #define TPMS_SESSION_AUDIT_INFO_Marshal(source, buffer, size)                           \
984         Marshal(TPMS_SESSION_AUDIT_INFO_MARSHAL_REF, (source), (buffer), (size))
985     #define TPMS_CREATION_INFO_Marshal(source, buffer, size)                                \
986         Marshal(TPMS_CREATION_INFO_MARSHAL_REF, (source), (buffer), (size))
987     #define TPMS_NV_CERTIFY_INFO_Marshal(source, buffer, size)                              \
988         Marshal(TPMS_NV_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
989     #define TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(source, buffer, size)                       \
990         Marshal(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
991     #define TPMI_ST_ATTEST_Marshal(source, buffer, size)                                    \
992         Marshal(TPMI_ST_ATTEST_MARSHAL_REF, (source), (buffer), (size))
993     #define TPMU_ATTEST_Marshal(source, buffer, size, selector)                             \
994         MarshalUnion(TPMU_ATTEST_MARSHAL_REF, (target), (buffer), (size), (selector))
995     #define TPMS_ATTEST_Marshal(source, buffer, size)                                       \
996         Marshal(TPMS_ATTEST_MARSHAL_REF, (source), (buffer), (size))
997     #define TPM2B_ATTEST_Marshal(source, buffer, size)                                      \
998         Marshal(TPM2B_ATTEST_MARSHAL_REF, (source), (buffer), (size))
999     #define TPMS_AUTH_COMMAND_Unmarshal(target, buffer, size)                               \
1000        Unmarshal(TPMS_AUTH_COMMAND_MARSHAL_REF, (target), (buffer), (size))
1001    #define TPMS_AUTH_RESPONSE_Marshal(source, buffer, size)                                \
1002        Marshal(TPMS_AUTH_RESPONSE_MARSHAL_REF, (source), (buffer), (size))
1003    #define TPMI_TDES_KEY_BITS_Unmarshal(target, buffer, size)                              \
1004        Unmarshal(TPMI_TDES_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1005    #define TPMI_TDES_KEY_BITS_Marshal(source, buffer, size)                                \
1006        Marshal(TPMI_TDES_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1007    #define TPMI_AES_KEY_BITS_Unmarshal(target, buffer, size)                               \
1008        Unmarshal(TPMI_AES_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1009    #define TPMI_AES_KEY_BITS_Marshal(source, buffer, size)                                 \
1010        Marshal(TPMI_AES_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1011    #define TPMI_SM4_KEY_BITS_Unmarshal(target, buffer, size)                               \
1012        Unmarshal(TPMI_SM4_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1013    #define TPMI_SM4_KEY_BITS_Marshal(source, buffer, size)                                 \
1014        Marshal(TPMI_SM4_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1015    #define TPMI_CAMELLIA_KEY_BITS_Unmarshal(target, buffer, size)                          \
1016        Unmarshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1017    #define TPMI_CAMELLIA_KEY_BITS_Marshal(source, buffer, size)                            \
1018        Marshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1019    #define TPMU_SYM_KEY_BITS_Unmarshal(target, buffer, size, selector)                     \
1020        UnmarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size),       \
1021          (selector))
1022    #define TPMU_SYM_KEY_BITS_Marshal(source, buffer, size, selector)                       \
1023        MarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size),         \
1024          (selector))
1025    #define TPMU_SYM_MODE_Unmarshal(target, buffer, size, selector)                         \
1026        UnmarshalUnion(TPMU_SYM_MODE_MARSHAL_REF, (target), (buffer), (size),           \
1027          (selector))
1028    #define TPMU_SYM_MODE_Marshal(source, buffer, size, selector)                           \
1029        MarshalUnion(TPMU_SYM_MODE_MARSHAL_REF, (target), (buffer), (size), (selector))
1030    #define TPMT_SYM_DEF_Unmarshal(target, buffer, size, flag)                              \
1031        Unmarshal(TPMT_SYM_DEF_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),  \
1032          (size))
1033    #define TPMT_SYM_DEF_Marshal(source, buffer, size)                                      \
```

```
1034        Marshal(TPMT_SYM_DEF_MARSHAL_REF, (source), (buffer), (size))
1035    #define TPMT_SYM_DEF_OBJECT_Unmarshal(target, buffer, size, flag)                    \
1036        Unmarshal(TPMT_SYM_DEF_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),      \
1037          (buffer), (size))
1038    #define TPMT_SYM_DEF_OBJECT_Marshal(source, buffer, size)                            \
1039        Marshal(TPMT_SYM_DEF_OBJECT_MARSHAL_REF, (source), (buffer), (size))
1040    #define TPM2B_SYM_KEY_Unmarshal(target, buffer, size)                                \
1041        Unmarshal(TPM2B_SYM_KEY_MARSHAL_REF, (target), (buffer), (size))
1042    #define TPM2B_SYM_KEY_Marshal(source, buffer, size)                                  \
1043        Marshal(TPM2B_SYM_KEY_MARSHAL_REF, (source), (buffer), (size))
1044    #define TPMS_SYMCIPHER_PARMS_Unmarshal(target, buffer, size)                         \
1045        Unmarshal(TPMS_SYMCIPHER_PARMS_MARSHAL_REF, (target), (buffer), (size))
1046    #define TPMS_SYMCIPHER_PARMS_Marshal(source, buffer, size)                           \
1047        Marshal(TPMS_SYMCIPHER_PARMS_MARSHAL_REF, (source), (buffer), (size))
1048    #define TPM2B_LABEL_Unmarshal(target, buffer, size)                                  \
1049        Unmarshal(TPM2B_LABEL_MARSHAL_REF, (target), (buffer), (size))
1050    #define TPM2B_LABEL_Marshal(source, buffer, size)                                    \
1051        Marshal(TPM2B_LABEL_MARSHAL_REF, (source), (buffer), (size))
1052    #define TPMS_DERIVE_Unmarshal(target, buffer, size)                                  \
1053        Unmarshal(TPMS_DERIVE_MARSHAL_REF, (target), (buffer), (size))
1054    #define TPMS_DERIVE_Marshal(source, buffer, size)                                    \
1055        Marshal(TPMS_DERIVE_MARSHAL_REF, (source), (buffer), (size))
1056    #define TPM2B_DERIVE_Unmarshal(target, buffer, size)                                 \
1057        Unmarshal(TPM2B_DERIVE_MARSHAL_REF, (target), (buffer), (size))
1058    #define TPM2B_DERIVE_Marshal(source, buffer, size)                                   \
1059        Marshal(TPM2B_DERIVE_MARSHAL_REF, (source), (buffer), (size))
1060    #define TPM2B_SENSITIVE_DATA_Unmarshal(target, buffer, size)                         \
1061        Unmarshal(TPM2B_SENSITIVE_DATA_MARSHAL_REF, (target), (buffer), (size))
1062    #define TPM2B_SENSITIVE_DATA_Marshal(source, buffer, size)                           \
1063        Marshal(TPM2B_SENSITIVE_DATA_MARSHAL_REF, (source), (buffer), (size))
1064    #define TPMS_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                        \
1065        Unmarshal(TPMS_SENSITIVE_CREATE_MARSHAL_REF, (target), (buffer), (size))
1066    #define TPM2B_SENSITIVE_CREATE_Unmarshal(target, buffer, size)                       \
1067        Unmarshal(TPM2B_SENSITIVE_CREATE_MARSHAL_REF, (target), (buffer), (size))
1068    #define TPMS_SCHEME_HASH_Unmarshal(target, buffer, size)                             \
1069        Unmarshal(TPMS_SCHEME_HASH_MARSHAL_REF, (target), (buffer), (size))
1070    #define TPMS_SCHEME_HASH_Marshal(source, buffer, size)                               \
1071        Marshal(TPMS_SCHEME_HASH_MARSHAL_REF, (source), (buffer), (size))
1072    #define TPMS_SCHEME_ECDAA_Unmarshal(target, buffer, size)                            \
1073        Unmarshal(TPMS_SCHEME_ECDAA_MARSHAL_REF, (target), (buffer), (size))
1074    #define TPMS_SCHEME_ECDAA_Marshal(source, buffer, size)                              \
1075        Marshal(TPMS_SCHEME_ECDAA_MARSHAL_REF, (source), (buffer), (size))
1076    #define TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)              \
1077        Unmarshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0),      \
1078          (target), (buffer), (size))
1079    #define TPMI_ALG_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                      \
1080        Marshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1081    #define TPMS_SCHEME_HMAC_Unmarshal(target, buffer, size)                             \
1082        Unmarshal(TPMS_SCHEME_HMAC_MARSHAL_REF, (target), (buffer), (size))
1083    #define TPMS_SCHEME_HMAC_Marshal(source, buffer, size)                               \
1084        Marshal(TPMS_SCHEME_HMAC_MARSHAL_REF, (source), (buffer), (size))
1085    #define TPMS_SCHEME_XOR_Unmarshal(target, buffer, size)                              \
1086        Unmarshal(TPMS_SCHEME_XOR_MARSHAL_REF, (target), (buffer), (size))
1087    #define TPMS_SCHEME_XOR_Marshal(source, buffer, size)                                \
1088        Marshal(TPMS_SCHEME_XOR_MARSHAL_REF, (source), (buffer), (size))
1089    #define TPMU_SCHEME_KEYEDHASH_Unmarshal(target, buffer, size, selector)              \
1090        UnmarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_REF, (target), (buffer), (size),   \
1091          (selector))
1092    #define TPMU_SCHEME_KEYEDHASH_Marshal(source, buffer, size, selector)                \
1093        MarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_REF, (target), (buffer), (size),     \
1094          (selector))
1095    #define TPMT_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag)                  \
1096        Unmarshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),   \
1097          (buffer), (size))
1098    #define TPMT_KEYEDHASH_SCHEME_Marshal(source, buffer, size)                          \
1099        Marshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_REF, (source), (buffer), (size))
```

```
1100  #define TPMS_SIG_SCHEME_RSASSA_Unmarshal(target, buffer, size)                    \
1101      Unmarshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF, (target), (buffer), (size))
1102  #define TPMS_SIG_SCHEME_RSASSA_Marshal(source, buffer, size)                      \
1103      Marshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF, (source), (buffer), (size))
1104  #define TPMS_SIG_SCHEME_RSAPSS_Unmarshal(target, buffer, size)                    \
1105      Unmarshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF, (target), (buffer), (size))
1106  #define TPMS_SIG_SCHEME_RSAPSS_Marshal(source, buffer, size)                      \
1107      Marshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF, (source), (buffer), (size))
1108  #define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size)                     \
1109      Unmarshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1110  #define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size)                       \
1111      Marshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1112  #define TPMS_SIG_SCHEME_SM2_Unmarshal(target, buffer, size)                       \
1113      Unmarshal(TPMS_SIG_SCHEME_SM2_MARSHAL_REF, (target), (buffer), (size))
1114  #define TPMS_SIG_SCHEME_SM2_Marshal(source, buffer, size)                         \
1115      Marshal(TPMS_SIG_SCHEME_SM2_MARSHAL_REF, (source), (buffer), (size))
1116  #define TPMS_SIG_SCHEME_ECSCHNORR_Unmarshal(target, buffer, size)                 \
1117      Unmarshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF, (target), (buffer), (size))
1118  #define TPMS_SIG_SCHEME_ECSCHNORR_Marshal(source, buffer, size)                   \
1119      Marshal(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF, (source), (buffer), (size))
1120  #define TPMS_SIG_SCHEME_ECDAA_Unmarshal(target, buffer, size)                     \
1121      Unmarshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF, (target), (buffer), (size))
1122  #define TPMS_SIG_SCHEME_ECDAA_Marshal(source, buffer, size)                       \
1123      Marshal(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF, (source), (buffer), (size))
1124  #define TPMU_SIG_SCHEME_Unmarshal(target, buffer, size, selector)                 \
1125      UnmarshalUnion(TPMU_SIG_SCHEME_MARSHAL_REF, (target), (buffer), (size),       \
1126        (selector))
1127  #define TPMU_SIG_SCHEME_Marshal(source, buffer, size, selector)                   \
1128      MarshalUnion(TPMU_SIG_SCHEME_MARSHAL_REF, (target), (buffer), (size),         \
1129        (selector))
1130  #define TPMT_SIG_SCHEME_Unmarshal(target, buffer, size, flag)                     \
1131      Unmarshal(TPMT_SIG_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),       \
1132        (buffer), (size))
1133  #define TPMT_SIG_SCHEME_Marshal(source, buffer, size)                             \
1134      Marshal(TPMT_SIG_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1135  #define TPMS_ENC_SCHEME_OAEP_Unmarshal(target, buffer, size)                      \
1136      Unmarshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_REF, (target), (buffer), (size))
1137  #define TPMS_ENC_SCHEME_OAEP_Marshal(source, buffer, size)                        \
1138      Marshal(TPMS_ENC_SCHEME_OAEP_MARSHAL_REF, (source), (buffer), (size))
1139  #define TPMS_ENC_SCHEME_RSAES_Unmarshal(target, buffer, size)                     \
1140      Unmarshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_REF, (target), (buffer), (size))
1141  #define TPMS_ENC_SCHEME_RSAES_Marshal(source, buffer, size)                       \
1142      Marshal(TPMS_ENC_SCHEME_RSAES_MARSHAL_REF, (source), (buffer), (size))
1143  #define TPMS_KEY_SCHEME_ECDH_Unmarshal(target, buffer, size)                      \
1144      Unmarshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_REF, (target), (buffer), (size))
1145  #define TPMS_KEY_SCHEME_ECDH_Marshal(source, buffer, size)                        \
1146      Marshal(TPMS_KEY_SCHEME_ECDH_MARSHAL_REF, (source), (buffer), (size))
1147  #define TPMS_KEY_SCHEME_ECMQV_Unmarshal(target, buffer, size)                     \
1148      Unmarshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF, (target), (buffer), (size))
1149  #define TPMS_KEY_SCHEME_ECMQV_Marshal(source, buffer, size)                       \
1150      Marshal(TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF, (source), (buffer), (size))
1151  #define TPMS_SCHEME_MGF1_Unmarshal(target, buffer, size)                          \
1152      Unmarshal(TPMS_SCHEME_MGF1_MARSHAL_REF, (target), (buffer), (size))
1153  #define TPMS_SCHEME_MGF1_Marshal(source, buffer, size)                            \
1154      Marshal(TPMS_SCHEME_MGF1_MARSHAL_REF, (source), (buffer), (size))
1155  #define TPMS_SCHEME_KDF1_SP800_56A_Unmarshal(target, buffer, size)                \
1156      Unmarshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF, (target), (buffer), (size))
1157  #define TPMS_SCHEME_KDF1_SP800_56A_Marshal(source, buffer, size)                  \
1158      Marshal(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF, (source), (buffer), (size))
1159  #define TPMS_SCHEME_KDF2_Unmarshal(target, buffer, size)                          \
1160      Unmarshal(TPMS_SCHEME_KDF2_MARSHAL_REF, (target), (buffer), (size))
1161  #define TPMS_SCHEME_KDF2_Marshal(source, buffer, size)                            \
1162      Marshal(TPMS_SCHEME_KDF2_MARSHAL_REF, (source), (buffer), (size))
1163  #define TPMS_SCHEME_KDF1_SP800_108_Unmarshal(target, buffer, size)                \
1164      Unmarshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF, (target), (buffer), (size))
1165  #define TPMS_SCHEME_KDF1_SP800_108_Marshal(source, buffer, size)                  \
```

```
1166        Marshal(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF, (source), (buffer), (size))
1167    #define TPMU_KDF_SCHEME_Unmarshal(target, buffer, size, selector)                   \
1168        UnmarshalUnion(TPMU_KDF_SCHEME_MARSHAL_REF, (target), (buffer), (size),         \
1169          (selector))
1170    #define TPMU_KDF_SCHEME_Marshal(source, buffer, size, selector)                     \
1171        MarshalUnion(TPMU_KDF_SCHEME_MARSHAL_REF, (target), (buffer), (size),           \
1172          (selector))
1173    #define TPMT_KDF_SCHEME_Unmarshal(target, buffer, size, flag)                       \
1174        Unmarshal(TPMT_KDF_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),         \
1175          (buffer), (size))
1176    #define TPMT_KDF_SCHEME_Marshal(source, buffer, size)                               \
1177        Marshal(TPMT_KDF_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1178    #define TPMI_ALG_ASYM_SCHEME_Unmarshal(target, buffer, size, flag)                  \
1179        Unmarshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
1180          (buffer), (size))
1181    #define TPMI_ALG_ASYM_SCHEME_Marshal(source, buffer, size)                          \
1182        Marshal(TPMI_ALG_ASYM_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1183    #define TPMU_ASYM_SCHEME_Unmarshal(target, buffer, size, selector)                  \
1184        UnmarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_REF, (target), (buffer), (size),        \
1185          (selector))
1186    #define TPMU_ASYM_SCHEME_Marshal(source, buffer, size, selector)                    \
1187        MarshalUnion(TPMU_ASYM_SCHEME_MARSHAL_REF, (target), (buffer), (size),          \
1188          (selector))
1189    #define TPMI_ALG_RSA_SCHEME_Unmarshal(target, buffer, size, flag)                   \
1190        Unmarshal(TPMI_ALG_RSA_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
1191          (buffer), (size))
1192    #define TPMI_ALG_RSA_SCHEME_Marshal(source, buffer, size)                           \
1193        Marshal(TPMI_ALG_RSA_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1194    #define TPMT_RSA_SCHEME_Unmarshal(target, buffer, size, flag)                       \
1195        Unmarshal(TPMT_RSA_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),         \
1196          (buffer), (size))
1197    #define TPMT_RSA_SCHEME_Marshal(source, buffer, size)                               \
1198        Marshal(TPMT_RSA_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1199    #define TPMI_ALG_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)                  \
1200        Unmarshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),    \
1201          (buffer), (size))
1202    #define TPMI_ALG_RSA_DECRYPT_Marshal(source, buffer, size)                          \
1203        Marshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_REF, (source), (buffer), (size))
1204    #define TPMT_RSA_DECRYPT_Unmarshal(target, buffer, size, flag)                      \
1205        Unmarshal(TPMT_RSA_DECRYPT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),        \
1206          (buffer), (size))
1207    #define TPMT_RSA_DECRYPT_Marshal(source, buffer, size)                              \
1208        Marshal(TPMT_RSA_DECRYPT_MARSHAL_REF, (source), (buffer), (size))
1209    #define TPM2B_PUBLIC_KEY_RSA_Unmarshal(target, buffer, size)                        \
1210        Unmarshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF, (target), (buffer), (size))
1211    #define TPM2B_PUBLIC_KEY_RSA_Marshal(source, buffer, size)                          \
1212        Marshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF, (source), (buffer), (size))
1213    #define TPMI_RSA_KEY_BITS_Unmarshal(target, buffer, size)                           \
1214        Unmarshal(TPMI_RSA_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1215    #define TPMI_RSA_KEY_BITS_Marshal(source, buffer, size)                             \
1216        Marshal(TPMI_RSA_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1217    #define TPM2B_PRIVATE_KEY_RSA_Unmarshal(target, buffer, size)                       \
1218        Unmarshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF, (target), (buffer), (size))
1219    #define TPM2B_PRIVATE_KEY_RSA_Marshal(source, buffer, size)                         \
1220        Marshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF, (source), (buffer), (size))
1221    #define TPM2B_ECC_PARAMETER_Unmarshal(target, buffer, size)                         \
1222        Unmarshal(TPM2B_ECC_PARAMETER_MARSHAL_REF, (target), (buffer), (size))
1223    #define TPM2B_ECC_PARAMETER_Marshal(source, buffer, size)                           \
1224        Marshal(TPM2B_ECC_PARAMETER_MARSHAL_REF, (source), (buffer), (size))
1225    #define TPMS_ECC_POINT_Unmarshal(target, buffer, size)                              \
1226        Unmarshal(TPMS_ECC_POINT_MARSHAL_REF, (target), (buffer), (size))
1227    #define TPMS_ECC_POINT_Marshal(source, buffer, size)                                \
1228        Marshal(TPMS_ECC_POINT_MARSHAL_REF, (source), (buffer), (size))
1229    #define TPM2B_ECC_POINT_Unmarshal(target, buffer, size)                             \
1230        Unmarshal(TPM2B_ECC_POINT_MARSHAL_REF, (target), (buffer), (size))
1231    #define TPM2B_ECC_POINT_Marshal(source, buffer, size)                               \
```

```
1232        Marshal(TPM2B_ECC_POINT_MARSHAL_REF, (source), (buffer), (size))
1233    #define TPMI_ALG_ECC_SCHEME_Unmarshal(target, buffer, size, flag)                    \
1234        Unmarshal(TPMI_ALG_ECC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),     \
1235          (buffer), (size))
1236    #define TPMI_ALG_ECC_SCHEME_Marshal(source, buffer, size)                            \
1237        Marshal(TPMI_ALG_ECC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1238    #define TPMI_ECC_CURVE_Unmarshal(target, buffer, size)                               \
1239        Unmarshal(TPMI_ECC_CURVE_MARSHAL_REF, (target), (buffer), (size))
1240    #define TPMI_ECC_CURVE_Marshal(source, buffer, size)                                 \
1241        Marshal(TPMI_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
1242    #define TPMT_ECC_SCHEME_Unmarshal(target, buffer, size, flag)                        \
1243        Unmarshal(TPMT_ECC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target),          \
1244          (buffer), (size))
1245    #define TPMT_ECC_SCHEME_Marshal(source, buffer, size)                                \
1246        Marshal(TPMT_ECC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1247    #define TPMS_ALGORITHM_DETAIL_ECC_Marshal(source, buffer, size)                      \
1248        Marshal(TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_REF, (source), (buffer), (size))
1249    #define TPMS_SIGNATURE_RSA_Unmarshal(target, buffer, size)                           \
1250        Unmarshal(TPMS_SIGNATURE_RSA_MARSHAL_REF, (target), (buffer), (size))
1251    #define TPMS_SIGNATURE_RSA_Marshal(source, buffer, size)                             \
1252        Marshal(TPMS_SIGNATURE_RSA_MARSHAL_REF, (source), (buffer), (size))
1253    #define TPMS_SIGNATURE_RSASSA_Unmarshal(target, buffer, size)                        \
1254        Unmarshal(TPMS_SIGNATURE_RSASSA_MARSHAL_REF, (target), (buffer), (size))
1255    #define TPMS_SIGNATURE_RSASSA_Marshal(source, buffer, size)                          \
1256        Marshal(TPMS_SIGNATURE_RSASSA_MARSHAL_REF, (source), (buffer), (size))
1257    #define TPMS_SIGNATURE_RSAPSS_Unmarshal(target, buffer, size)                        \
1258        Unmarshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF, (target), (buffer), (size))
1259    #define TPMS_SIGNATURE_RSAPSS_Marshal(source, buffer, size)                          \
1260        Marshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF, (source), (buffer), (size))
1261    #define TPMS_SIGNATURE_ECC_Unmarshal(target, buffer, size)                           \
1262        Unmarshal(TPMS_SIGNATURE_ECC_MARSHAL_REF, (target), (buffer), (size))
1263    #define TPMS_SIGNATURE_ECC_Marshal(source, buffer, size)                             \
1264        Marshal(TPMS_SIGNATURE_ECC_MARSHAL_REF, (source), (buffer), (size))
1265    #define TPMS_SIGNATURE_ECDAA_Unmarshal(target, buffer, size)                         \
1266        Unmarshal(TPMS_SIGNATURE_ECDAA_MARSHAL_REF, (target), (buffer), (size))
1267    #define TPMS_SIGNATURE_ECDAA_Marshal(source, buffer, size)                           \
1268        Marshal(TPMS_SIGNATURE_ECDAA_MARSHAL_REF, (source), (buffer), (size))
1269    #define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size)                         \
1270        Unmarshal(TPMS_SIGNATURE_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1271    #define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size)                           \
1272        Marshal(TPMS_SIGNATURE_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1273    #define TPMS_SIGNATURE_SM2_Unmarshal(target, buffer, size)                           \
1274        Unmarshal(TPMS_SIGNATURE_SM2_MARSHAL_REF, (target), (buffer), (size))
1275    #define TPMS_SIGNATURE_SM2_Marshal(source, buffer, size)                             \
1276        Marshal(TPMS_SIGNATURE_SM2_MARSHAL_REF, (source), (buffer), (size))
1277    #define TPMS_SIGNATURE_ECSCHNORR_Unmarshal(target, buffer, size)                     \
1278        Unmarshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF, (target), (buffer), (size))
1279    #define TPMS_SIGNATURE_ECSCHNORR_Marshal(source, buffer, size)                       \
1280        Marshal(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF, (source), (buffer), (size))
1281    #define TPMU_SIGNATURE_Unmarshal(target, buffer, size, selector)                     \
1282        UnmarshalUnion(TPMU_SIGNATURE_MARSHAL_REF, (target), (buffer), (size),           \
1283          (selector))
1284    #define TPMU_SIGNATURE_Marshal(source, buffer, size, selector)                       \
1285        MarshalUnion(TPMU_SIGNATURE_MARSHAL_REF, (target), (buffer), (size), (selector))
1286    #define TPMT_SIGNATURE_Unmarshal(target, buffer, size, flag)                         \
1287        Unmarshal(TPMT_SIGNATURE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),\
1288          (size))
1289    #define TPMT_SIGNATURE_Marshal(source, buffer, size)                                 \
1290        Marshal(TPMT_SIGNATURE_MARSHAL_REF, (source), (buffer), (size))
1291    #define TPMU_ENCRYPTED_SECRET_Unmarshal(target, buffer, size, selector)              \
1292        UnmarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size),    \
1293          (selector))
1294    #define TPMU_ENCRYPTED_SECRET_Marshal(source, buffer, size, selector)                \
1295        MarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size),      \
1296          (selector))
1297    #define TPM2B_ENCRYPTED_SECRET_Unmarshal(target, buffer, size)                       \
```

```
1298        Unmarshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size))
1299    #define TPM2B_ENCRYPTED_SECRET_Marshal(source, buffer, size)                        \
1300        Marshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_REF, (source), (buffer), (size))
1301    #define TPMI_ALG_PUBLIC_Unmarshal(target, buffer, size)                             \
1302        Unmarshal(TPMI_ALG_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1303    #define TPMI_ALG_PUBLIC_Marshal(source, buffer, size)                               \
1304        Marshal(TPMI_ALG_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1305    #define TPMU_PUBLIC_ID_Unmarshal(target, buffer, size, selector)                    \
1306        UnmarshalUnion(TPMU_PUBLIC_ID_MARSHAL_REF, (target), (buffer), (size),          \
1307          (selector))
1308    #define TPMU_PUBLIC_ID_Marshal(source, buffer, size, selector)                      \
1309        MarshalUnion(TPMU_PUBLIC_ID_MARSHAL_REF, (target), (buffer), (size), (selector))
1310    #define TPMS_KEYEDHASH_PARMS_Unmarshal(target, buffer, size)                        \
1311        Unmarshal(TPMS_KEYEDHASH_PARMS_MARSHAL_REF, (target), (buffer), (size))
1312    #define TPMS_KEYEDHASH_PARMS_Marshal(source, buffer, size)                          \
1313        Marshal(TPMS_KEYEDHASH_PARMS_MARSHAL_REF, (source), (buffer), (size))
1314    #define TPMS_RSA_PARMS_Unmarshal(target, buffer, size)                              \
1315        Unmarshal(TPMS_RSA_PARMS_MARSHAL_REF, (target), (buffer), (size))
1316    #define TPMS_RSA_PARMS_Marshal(source, buffer, size)                                \
1317        Marshal(TPMS_RSA_PARMS_MARSHAL_REF, (source), (buffer), (size))
1318    #define TPMS_ECC_PARMS_Unmarshal(target, buffer, size)                              \
1319        Unmarshal(TPMS_ECC_PARMS_MARSHAL_REF, (target), (buffer), (size))
1320    #define TPMS_ECC_PARMS_Marshal(source, buffer, size)                                \
1321        Marshal(TPMS_ECC_PARMS_MARSHAL_REF, (source), (buffer), (size))
1322    #define TPMU_PUBLIC_PARMS_Unmarshal(target, buffer, size, selector)                 \
1323        UnmarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size),       \
1324          (selector))
1325    #define TPMU_PUBLIC_PARMS_Marshal(source, buffer, size, selector)                   \
1326        MarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size),         \
1327          (selector))
1328    #define TPMT_PUBLIC_PARMS_Unmarshal(target, buffer, size)                           \
1329        Unmarshal(TPMT_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size))
1330    #define TPMT_PUBLIC_PARMS_Marshal(source, buffer, size)                             \
1331        Marshal(TPMT_PUBLIC_PARMS_MARSHAL_REF, (source), (buffer), (size))
1332    #define TPMT_PUBLIC_Unmarshal(target, buffer, size, flag)                           \
1333        Unmarshal(TPMT_PUBLIC_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),   \
1334          (size))
1335    #define TPMT_PUBLIC_Marshal(source, buffer, size)                                   \
1336        Marshal(TPMT_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1337    #define TPM2B_PUBLIC_Unmarshal(target, buffer, size, flag)                          \
1338        Unmarshal(TPM2B_PUBLIC_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer),  \
1339          (size))
1340    #define TPM2B_PUBLIC_Marshal(source, buffer, size)                                  \
1341        Marshal(TPM2B_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1342    #define TPM2B_TEMPLATE_Unmarshal(target, buffer, size)                              \
1343        Unmarshal(TPM2B_TEMPLATE_MARSHAL_REF, (target), (buffer), (size))
1344    #define TPM2B_TEMPLATE_Marshal(source, buffer, size)                                \
1345        Marshal(TPM2B_TEMPLATE_MARSHAL_REF, (source), (buffer), (size))
1346    #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(target, buffer, size)               \
1347        Unmarshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF, (target), (buffer), (size))
1348    #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(source, buffer, size)                 \
1349        Marshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF, (source), (buffer), (size))
1350    #define TPMU_SENSITIVE_COMPOSITE_Unmarshal(target, buffer, size, selector)          \
1351        UnmarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF, (target), (buffer), (size),\
1352          (selector))
1353    #define TPMU_SENSITIVE_COMPOSITE_Marshal(source, buffer, size, selector)            \
1354        MarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF, (target), (buffer), (size),  \
1355          (selector))
1356    #define TPMT_SENSITIVE_Unmarshal(target, buffer, size)                              \
1357        Unmarshal(TPMT_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1358    #define TPMT_SENSITIVE_Marshal(source, buffer, size)                                \
1359        Marshal(TPMT_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1360    #define TPM2B_SENSITIVE_Unmarshal(target, buffer, size)                             \
1361        Unmarshal(TPM2B_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1362    #define TPM2B_SENSITIVE_Marshal(source, buffer, size)                               \
1363        Marshal(TPM2B_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
```

```
1364  #define TPM2B_PRIVATE_Unmarshal(target, buffer, size)                                   \
1365      Unmarshal(TPM2B_PRIVATE_MARSHAL_REF, (target), (buffer), (size))
1366  #define TPM2B_PRIVATE_Marshal(source, buffer, size)                                     \
1367      Marshal(TPM2B_PRIVATE_MARSHAL_REF, (source), (buffer), (size))
1368  #define TPM2B_ID_OBJECT_Unmarshal(target, buffer, size)                                 \
1369      Unmarshal(TPM2B_ID_OBJECT_MARSHAL_REF, (target), (buffer), (size))
1370  #define TPM2B_ID_OBJECT_Marshal(source, buffer, size)                                   \
1371      Marshal(TPM2B_ID_OBJECT_MARSHAL_REF, (source), (buffer), (size))
1372  #define TPM_NV_INDEX_Marshal(source, buffer, size)                                      \
1373      Marshal(TPM_NV_INDEX_MARSHAL_REF, (source), (buffer), (size))
1374  #define TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(target, buffer, size)                  \
1375      Unmarshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF, (target), (buffer),        \
1376        (size))
1377  #define TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(source, buffer, size)                    \
1378      Marshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF, (source), (buffer), (size))
1379  #define TPMA_NV_Unmarshal(target, buffer, size)                                         \
1380      Unmarshal(TPMA_NV_MARSHAL_REF, (target), (buffer), (size))
1381  #define TPMA_NV_Marshal(source, buffer, size)                                           \
1382      Marshal(TPMA_NV_MARSHAL_REF, (source), (buffer), (size))
1383  #define TPMS_NV_PUBLIC_Unmarshal(target, buffer, size)                                  \
1384      Unmarshal(TPMS_NV_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1385  #define TPMS_NV_PUBLIC_Marshal(source, buffer, size)                                    \
1386      Marshal(TPMS_NV_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1387  #define TPM2B_NV_PUBLIC_Unmarshal(target, buffer, size)                                 \
1388      Unmarshal(TPM2B_NV_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1389  #define TPM2B_NV_PUBLIC_Marshal(source, buffer, size)                                   \
1390      Marshal(TPM2B_NV_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1391  #define TPM2B_CONTEXT_SENSITIVE_Unmarshal(target, buffer, size)                         \
1392      Unmarshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1393  #define TPM2B_CONTEXT_SENSITIVE_Marshal(source, buffer, size)                           \
1394      Marshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1395  #define TPMS_CONTEXT_DATA_Unmarshal(target, buffer, size)                               \
1396      Unmarshal(TPMS_CONTEXT_DATA_MARSHAL_REF, (target), (buffer), (size))
1397  #define TPMS_CONTEXT_DATA_Marshal(source, buffer, size)                                 \
1398      Marshal(TPMS_CONTEXT_DATA_MARSHAL_REF, (source), (buffer), (size))
1399  #define TPM2B_CONTEXT_DATA_Unmarshal(target, buffer, size)                              \
1400      Unmarshal(TPM2B_CONTEXT_DATA_MARSHAL_REF, (target), (buffer), (size))
1401  #define TPM2B_CONTEXT_DATA_Marshal(source, buffer, size)                                \
1402      Marshal(TPM2B_CONTEXT_DATA_MARSHAL_REF, (source), (buffer), (size))
1403  #define TPMS_CONTEXT_Unmarshal(target, buffer, size)                                    \
1404      Unmarshal(TPMS_CONTEXT_MARSHAL_REF, (target), (buffer), (size))
1405  #define TPMS_CONTEXT_Marshal(source, buffer, size)                                      \
1406      Marshal(TPMS_CONTEXT_MARSHAL_REF, (source), (buffer), (size))
1407  #define TPMS_CREATION_DATA_Marshal(source, buffer, size)                                \
1408      Marshal(TPMS_CREATION_DATA_MARSHAL_REF, (source), (buffer), (size))
1409  #define TPM2B_CREATION_DATA_Marshal(source, buffer, size)                               \
1410      Marshal(TPM2B_CREATION_DATA_MARSHAL_REF, (source), (buffer), (size))
1411  #define TPM_AT_Unmarshal(target, buffer, size)                                          \
1412      Unmarshal(TPM_AT_MARSHAL_REF, (target), (buffer), (size))
1413  #define TPM_AT_Marshal(source, buffer, size)                                            \
1414      Marshal(TPM_AT_MARSHAL_REF, (source), (buffer), (size))
1415  #define TPM_AE_Marshal(source, buffer, size)                                            \
1416      Marshal(TPM_AE_MARSHAL_REF, (source), (buffer), (size))
1417  #define TPMS_AC_OUTPUT_Marshal(source, buffer, size)                                    \
1418      Marshal(TPMS_AC_OUTPUT_MARSHAL_REF, (source), (buffer), (size))
1419  #define TPML_AC_CAPABILITIES_Marshal(source, buffer, size)                              \
1420      Marshal(TPML_AC_CAPABILITIES_MARSHAL_REF, (source), (buffer), (size))
1421  #endif // _TABLE_MARSHAL_DEFINES_H_
```

9.10.7.4 TableMarshalTypes.h

```
1  #ifndef _TABLE_MARSHAL_TYPES_H_
2  #define _TABLE_MARSHAL_TYPES_H_
3  typedef UINT16      marshalIndex_t;
```

9.10.7.4.1.1 Structure Entries

A structure contains a list of elements to unmarshal. Each of the entries is a UINT16. The structure descriptor is: The *values* array contains indicators for the things to marshal. The *elements* parameter indicates how many different entities are unmarshaled. This number nominally corresponds to the number of rows in the Part 2 table that describes the structure (the number of rows minus the title row and any error code rows). A schematic of a simple structure entry is shown here but the values are not actually in a structure. As shown, the third value is the offset in the structure where the value is placed when unmarshaled, or fetched from when marshaling. This is sufficient when the element type indicated by *index* is always a simple type and never a union or array.This is just shown for illustrative purposes.

```
typedef struct simpleStructureEntry_t {
    UINT16          qualifiers;         // indicates the type of entry (array, union
                                        // etc.)
    marshalIndex_t  index;              // the index into the appropriate array of
                                        //  the descriptor of this type
    UINT16          offset;             // where this comes from or is placed
} simpleStructureEntry_t;
typedef const struct UintMarshal_mst
{
    UINT8           marshalType;        // UINT_MTYPE
    UINT8           modifiers;          // size and signed indicator.
} UintMarshal_mst;
typedef struct UnionMarshal_mst
{
    UINT8           countOfselectors;
    UINT8           modifiers;          // NULL_SELECTOR
    UINT16          offsetOfUnmarshalTypes;
    UINT32          selectors[1];
//    UINT16          marshalingTypes[1]; // This is not part of the prototypical
                                          //    entry. It is here to show where the
                                          //    marshaling types will be in a union
} UnionMarshal_mst;
typedef struct NullUnionMarshal_mst
{
    UINT8           count;
} NullUnionMarshal_mst;
typedef struct MarshalHeader_mst
{
    UINT8           marshalType;        // VALUES_MTYPE
    UINT8           modifiers;
    UINT8           errorCode;
} MarshalHeader_mst;
typedef const struct ArrayMarshal_mst   // used in a structure
{
    marshalIndex_t  type;
    UINT16          stride;
} ArrayMarshal_mst;
typedef const struct StructMarshal_mst
{
    UINT8           marshalType;        // STRUCTURE_MTYPE
    UINT8           elements;
    UINT16          values[1];          // three times elements
} StructMarshal_mst;
typedef const struct ValuesMarshal_mst
{
    UINT8           marshalType;        // VALUES_MTYPE
    UINT8           modifiers;
    UINT8           errorCode;
    UINT8           ranges;
    UINT8           singles;
    UINT32          values[1];
} ValuesMarshal_mst;
```



```
56  typedef const struct TableMarshal_mst
57  {
58      UINT8           marshalType;        // TABLE_MTYPE
59      UINT8           modifiers;
60      UINT8           errorCode;
61      UINT8           singles;
62      UINT32          values[1];
63  } TableMarshal_mst;
64  typedef const struct MinMaxMarshal_mst
65  {
66      UINT8           marshalType;        // MIN_MAX_MTYPE
67      UINT8           modifiers;
68      UINT8           errorCode;
69      UINT32          values[2];
70  } MinMaxMarshal_mst;
71  typedef const struct Tpm2bMarshal_mst
72  {
73      UINT8           unmarshalType;      // TPM2B_MTYPE
74      UINT16          sizeIndex;          // reference to type for this size value
75  } Tpm2bMarshal_mst;
76  typedef const struct Tpm2bsMarshal_mst
77  {
78      UINT8           unmarshalType;      // TPM2BS_MTYPE
79      UINT8           modifiers;          // size= and offset (2 - 7)
80      UINT16          sizeIndex;          // index of the size value;
81      UINT16          dataIndex;          // the structure
82  } Tpm2bsMarshal_mst;
83  typedef const struct ListMarshal_mst
84  {
85      UINT8           unmarshalType;      // LIST_MTYPE (for TPML)
86      UINT8           modifiers;          // size offset 2-7
87      UINT16          sizeIndex;          // reference to the minmax structure that
88                                          //      unmarshals the size parameter
89      UINT16          arrayRef;           // reference to an array definition (type
90                                          //  and stride)
91  } ListMarshal_mst;
92  typedef const struct AttributesMarshal_mst
93  {
94      UINT8           unmarashalType;     // ATTRIBUTE_MTYPE
95      UINT8           modifiers;          // size (ONE_BYTES, TWO_BYTES, or FOUR_BYTES
96      UINT32          attributeMask;      // the values that must be zero.
97  } AttributesMarshal_mst;
98  typedef const struct CompositeMarshal_mst
99  {
100     UINT8           unmashalType;       // COMPOSITE_MTYPE
101     UINT8           modifiers;          // number of entries and size
102     marshalIndex_t  types[1];           // array of unmarshaling types
103 } CompositeMarshal_mst;
104 typedef const struct TPM_ECC_CURVE_mst {
105     UINT8       marshalType;
106     UINT8       modifiers;
107     UINT8       errorCode;
108     UINT32      values[4];
109 } TPM_ECC_CURVE_mst;
110 typedef const struct TPM_CLOCK_ADJUST_mst {
111     UINT8       marshalType;
112     UINT8       modifiers;
113     UINT8       errorCode;
114     UINT32      values[2];
115 } TPM_CLOCK_ADJUST_mst;
116 typedef const struct TPM_EO_mst {
117     UINT8       marshalType;
118     UINT8       modifiers;
119     UINT8       errorCode;
120     UINT32      values[2];
121 } TPM_EO_mst;
```

```
122  typedef const struct TPM_SU_mst {
123      UINT8          marshalType;
124      UINT8          modifiers;
125      UINT8          errorCode;
126      UINT8          entries;
127      UINT32         values[2];
128  } TPM_SU_mst;
129  typedef const struct TPM_SE_mst {
130      UINT8          marshalType;
131      UINT8          modifiers;
132      UINT8          errorCode;
133      UINT8          entries;
134      UINT32         values[3];
135  } TPM_SE_mst;
136  typedef const struct TPM_CAP_mst {
137      UINT8          marshalType;
138      UINT8          modifiers;
139      UINT8          errorCode;
140      UINT8          ranges;
141      UINT8          singles;
142      UINT32         values[3];
143  } TPM_CAP_mst;
144  typedef const struct TPMI_YES_NO_mst {
145      UINT8          marshalType;
146      UINT8          modifiers;
147      UINT8          errorCode;
148      UINT8          entries;
149      UINT32         values[2];
150  } TPMI_YES_NO_mst;
151  typedef const struct TPMI_DH_OBJECT_mst {
152      UINT8          marshalType;
153      UINT8          modifiers;
154      UINT8          errorCode;
155      UINT8          ranges;
156      UINT8          singles;
157      UINT32         values[5];
158  } TPMI_DH_OBJECT_mst;
159  typedef const struct TPMI_DH_PARENT_mst {
160      UINT8          marshalType;
161      UINT8          modifiers;
162      UINT8          errorCode;
163      UINT8          ranges;
164      UINT8          singles;
165      UINT32         values[8];
166  } TPMI_DH_PARENT_mst;
167  typedef const struct TPMI_DH_PERSISTENT_mst {
168      UINT8        marshalType;
169      UINT8        modifiers;
170      UINT8        errorCode;
171      UINT32       values[2];
172  } TPMI_DH_PERSISTENT_mst;
173  typedef const struct TPMI_DH_ENTITY_mst {
174      UINT8          marshalType;
175      UINT8          modifiers;
176      UINT8          errorCode;
177      UINT8          ranges;
178      UINT8          singles;
179      UINT32         values[15];
180  } TPMI_DH_ENTITY_mst;
181  typedef const struct TPMI_DH_PCR_mst {
182      UINT8        marshalType;
183      UINT8        modifiers;
184      UINT8        errorCode;
185      UINT32       values[3];
186  } TPMI_DH_PCR_mst;
187  typedef const struct TPMI_SH_AUTH_SESSION_mst {
```

```
188         UINT8           marshalType;
189         UINT8           modifiers;
190         UINT8           errorCode;
191         UINT8           ranges;
192         UINT8           singles;
193         UINT32          values[5];
194     } TPMI_SH_AUTH_SESSION_mst;
195     typedef const struct TPMI_SH_HMAC_mst {
196         UINT8         marshalType;
197         UINT8         modifiers;
198         UINT8         errorCode;
199         UINT32        values[2];
200     } TPMI_SH_HMAC_mst;
201     typedef const struct TPMI_SH_POLICY_mst {
202         UINT8         marshalType;
203         UINT8         modifiers;
204         UINT8         errorCode;
205         UINT32        values[2];
206     } TPMI_SH_POLICY_mst;
207     typedef const struct TPMI_DH_CONTEXT_mst {
208         UINT8           marshalType;
209         UINT8           modifiers;
210         UINT8           errorCode;
211         UINT8           ranges;
212         UINT8           singles;
213         UINT32          values[6];
214     } TPMI_DH_CONTEXT_mst;
215     typedef const struct TPMI_DH_SAVED_mst {
216         UINT8           marshalType;
217         UINT8           modifiers;
218         UINT8           errorCode;
219         UINT8           ranges;
220         UINT8           singles;
221         UINT32          values[7];
222     } TPMI_DH_SAVED_mst;
223     typedef const struct TPMI_RH_HIERARCHY_mst {
224         UINT8           marshalType;
225         UINT8           modifiers;
226         UINT8           errorCode;
227         UINT8           entries;
228         UINT32          values[4];
229     } TPMI_RH_HIERARCHY_mst;
230     typedef const struct TPMI_RH_ENABLES_mst {
231         UINT8           marshalType;
232         UINT8           modifiers;
233         UINT8           errorCode;
234         UINT8           entries;
235         UINT32          values[5];
236     } TPMI_RH_ENABLES_mst;
237     typedef const struct TPMI_RH_HIERARCHY_AUTH_mst {
238         UINT8           marshalType;
239         UINT8           modifiers;
240         UINT8           errorCode;
241         UINT8           entries;
242         UINT32          values[4];
243     } TPMI_RH_HIERARCHY_AUTH_mst;
244     typedef const struct TPMI_RH_HIERARCHY_POLICY_mst {
245         UINT8           marshalType;
246         UINT8           modifiers;
247         UINT8           errorCode;
248         UINT8           ranges;
249         UINT8           singles;
250         UINT32          values[6];
251     } TPMI_RH_HIERARCHY_POLICY_mst;
252     typedef const struct TPMI_RH_PLATFORM_mst {
253         UINT8           marshalType;
```

```
254         UINT8           modifiers;
255         UINT8           errorCode;
256         UINT8           entries;
257         UINT32          values[1];
258     } TPMI_RH_PLATFORM_mst;
259     typedef const struct TPMI_RH_OWNER_mst {
260         UINT8           marshalType;
261         UINT8           modifiers;
262         UINT8           errorCode;
263         UINT8           entries;
264         UINT32          values[2];
265     } TPMI_RH_OWNER_mst;
266     typedef const struct TPMI_RH_ENDORSEMENT_mst {
267         UINT8           marshalType;
268         UINT8           modifiers;
269         UINT8           errorCode;
270         UINT8           entries;
271         UINT32          values[2];
272     } TPMI_RH_ENDORSEMENT_mst;
273     typedef const struct TPMI_RH_PROVISION_mst {
274         UINT8           marshalType;
275         UINT8           modifiers;
276         UINT8           errorCode;
277         UINT8           entries;
278         UINT32          values[2];
279     } TPMI_RH_PROVISION_mst;
280     typedef const struct TPMI_RH_CLEAR_mst {
281         UINT8           marshalType;
282         UINT8           modifiers;
283         UINT8           errorCode;
284         UINT8           entries;
285         UINT32          values[2];
286     } TPMI_RH_CLEAR_mst;
287     typedef const struct TPMI_RH_NV_AUTH_mst {
288         UINT8           marshalType;
289         UINT8           modifiers;
290         UINT8           errorCode;
291         UINT8           ranges;
292         UINT8           singles;
293         UINT32          values[4];
294     } TPMI_RH_NV_AUTH_mst;
295     typedef const struct TPMI_RH_LOCKOUT_mst {
296         UINT8           marshalType;
297         UINT8           modifiers;
298         UINT8           errorCode;
299         UINT8           entries;
300         UINT32          values[1];
301     } TPMI_RH_LOCKOUT_mst;
302     typedef const struct TPMI_RH_NV_INDEX_mst {
303         UINT8         marshalType;
304         UINT8         modifiers;
305         UINT8         errorCode;
306         UINT32        values[2];
307     } TPMI_RH_NV_INDEX_mst;
308     typedef const struct TPMI_RH_AC_mst {
309         UINT8         marshalType;
310         UINT8         modifiers;
311         UINT8         errorCode;
312         UINT32        values[2];
313     } TPMI_RH_AC_mst;
314     typedef const struct TPMI_RH_ACT_mst {
315         UINT8         marshalType;
316         UINT8         modifiers;
317         UINT8         errorCode;
318         UINT32        values[2];
319     } TPMI_RH_ACT_mst;
```

```
320   typedef const struct TPMI_ALG_HASH_mst {
321       UINT8       marshalType;
322       UINT8       modifiers;
323       UINT8       errorCode;
324       UINT32      values[5];
325   } TPMI_ALG_HASH_mst;
326   typedef const struct TPMI_ALG_ASYM_mst {
327       UINT8       marshalType;
328       UINT8       modifiers;
329       UINT8       errorCode;
330       UINT32      values[5];
331   } TPMI_ALG_ASYM_mst;
332   typedef const struct TPMI_ALG_SYM_mst {
333       UINT8       marshalType;
334       UINT8       modifiers;
335       UINT8       errorCode;
336       UINT32      values[5];
337   } TPMI_ALG_SYM_mst;
338   typedef const struct TPMI_ALG_SYM_OBJECT_mst {
339       UINT8       marshalType;
340       UINT8       modifiers;
341       UINT8       errorCode;
342       UINT32      values[5];
343   } TPMI_ALG_SYM_OBJECT_mst;
344   typedef const struct TPMI_ALG_SYM_MODE_mst {
345       UINT8       marshalType;
346       UINT8       modifiers;
347       UINT8       errorCode;
348       UINT32      values[4];
349   } TPMI_ALG_SYM_MODE_mst;
350   typedef const struct TPMI_ALG_KDF_mst {
351       UINT8       marshalType;
352       UINT8       modifiers;
353       UINT8       errorCode;
354       UINT32      values[4];
355   } TPMI_ALG_KDF_mst;
356   typedef const struct TPMI_ALG_SIG_SCHEME_mst {
357       UINT8       marshalType;
358       UINT8       modifiers;
359       UINT8       errorCode;
360       UINT32      values[4];
361   } TPMI_ALG_SIG_SCHEME_mst;
362   typedef const struct TPMI_ECC_KEY_EXCHANGE_mst {
363       UINT8       marshalType;
364       UINT8       modifiers;
365       UINT8       errorCode;
366       UINT32      values[4];
367   } TPMI_ECC_KEY_EXCHANGE_mst;
368   typedef const struct TPMI_ST_COMMAND_TAG_mst {
369       UINT8         marshalType;
370       UINT8         modifiers;
371       UINT8         errorCode;
372       UINT8         entries;
373       UINT32        values[2];
374   } TPMI_ST_COMMAND_TAG_mst;
375   typedef const struct TPMI_ALG_MAC_SCHEME_mst {
376       UINT8       marshalType;
377       UINT8       modifiers;
378       UINT8       errorCode;
379       UINT32      values[5];
380   } TPMI_ALG_MAC_SCHEME_mst;
381   typedef const struct TPMI_ALG_CIPHER_MODE_mst {
382       UINT8       marshalType;
383       UINT8       modifiers;
384       UINT8       errorCode;
385       UINT32      values[4];
```

```
386 } TPMI_ALG_CIPHER_MODE_mst;
387 typedef const struct TPMS_EMPTY_mst
388 {
389     UINT8     marshalType;
390     UINT8     elements;
391     UINT16    values[3];
392 } TPMS_EMPTY_mst;
393 typedef const struct TPMS_ALGORITHM_DESCRIPTION_mst
394 {
395     UINT8     marshalType;
396     UINT8     elements;
397     UINT16    values[6];
398 } TPMS_ALGORITHM_DESCRIPTION_mst;
399 typedef struct TPMU_HA_mst
400 {
401     BYTE            countOfselectors;
402     BYTE            modifiers;
403     UINT16        offsetOfUnmarshalTypes;
404     UINT32        selectors[9];
405     UINT16        marshalingTypes[9];
406 } TPMU_HA_mst;
407 typedef const struct TPMT_HA_mst
408 {
409     UINT8     marshalType;
410     UINT8     elements;
411     UINT16    values[6];
412 } TPMT_HA_mst;
413 typedef const struct TPMS_PCR_SELECT_mst
414 {
415     UINT8     marshalType;
416     UINT8     elements;
417     UINT16    values[6];
418 } TPMS_PCR_SELECT_mst;
419 typedef const struct TPMS_PCR_SELECTION_mst
420 {
421     UINT8     marshalType;
422     UINT8     elements;
423     UINT16    values[9];
424 } TPMS_PCR_SELECTION_mst;
425 typedef const struct TPMT_TK_CREATION_mst
426 {
427     UINT8     marshalType;
428     UINT8     elements;
429     UINT16    values[9];
430 } TPMT_TK_CREATION_mst;
431 typedef const struct TPMT_TK_VERIFIED_mst
432 {
433     UINT8     marshalType;
434     UINT8     elements;
435     UINT16    values[9];
436 } TPMT_TK_VERIFIED_mst;
437 typedef const struct TPMT_TK_AUTH_mst
438 {
439     UINT8     marshalType;
440     UINT8     elements;
441     UINT16    values[9];
442 } TPMT_TK_AUTH_mst;
443 typedef const struct TPMT_TK_HASHCHECK_mst
444 {
445     UINT8     marshalType;
446     UINT8     elements;
447     UINT16    values[9];
448 } TPMT_TK_HASHCHECK_mst;
449 typedef const struct TPMS_ALG_PROPERTY_mst
450 {
451     UINT8     marshalType;
```

```
452         UINT8     elements;
453         UINT16    values[6];
454     } TPMS_ALG_PROPERTY_mst;
455     typedef const struct TPMS_TAGGED_PROPERTY_mst
456     {
457         UINT8     marshalType;
458         UINT8     elements;
459         UINT16    values[6];
460     } TPMS_TAGGED_PROPERTY_mst;
461     typedef const struct TPMS_TAGGED_PCR_SELECT_mst
462     {
463         UINT8     marshalType;
464         UINT8     elements;
465         UINT16    values[9];
466     } TPMS_TAGGED_PCR_SELECT_mst;
467     typedef const struct TPMS_TAGGED_POLICY_mst
468     {
469         UINT8     marshalType;
470         UINT8     elements;
471         UINT16    values[6];
472     } TPMS_TAGGED_POLICY_mst;
473     typedef const struct TPMS_ACT_DATA_mst
474     {
475         UINT8     marshalType;
476         UINT8     elements;
477         UINT16    values[9];
478     } TPMS_ACT_DATA_mst;
479     typedef struct TPMU_CAPABILITIES_mst
480     {
481         BYTE            countOfselectors;
482         BYTE            modifiers;
483         UINT16        offsetOfUnmarshalTypes;
484         UINT32        selectors[11];
485         UINT16        marshalingTypes[11];
486     } TPMU_CAPABILITIES_mst;
487     typedef const struct TPMS_CAPABILITY_DATA_mst
488     {
489         UINT8     marshalType;
490         UINT8     elements;
491         UINT16    values[6];
492     } TPMS_CAPABILITY_DATA_mst;
493     typedef const struct TPMS_CLOCK_INFO_mst
494     {
495         UINT8     marshalType;
496         UINT8     elements;
497         UINT16    values[12];
498     } TPMS_CLOCK_INFO_mst;
499     typedef const struct TPMS_TIME_INFO_mst
500     {
501         UINT8     marshalType;
502         UINT8     elements;
503         UINT16    values[6];
504     } TPMS_TIME_INFO_mst;
505     typedef const struct TPMS_TIME_ATTEST_INFO_mst
506     {
507         UINT8     marshalType;
508         UINT8     elements;
509         UINT16    values[6];
510     } TPMS_TIME_ATTEST_INFO_mst;
511     typedef const struct TPMS_CERTIFY_INFO_mst
512     {
513         UINT8     marshalType;
514         UINT8     elements;
515         UINT16    values[6];
516     } TPMS_CERTIFY_INFO_mst;
517     typedef const struct TPMS_QUOTE_INFO_mst
```

```
518   {
519       UINT8     marshalType;
520       UINT8     elements;
521       UINT16    values[6];
522   } TPMS_QUOTE_INFO_mst;
523   typedef const struct TPMS_COMMAND_AUDIT_INFO_mst
524   {
525       UINT8     marshalType;
526       UINT8     elements;
527       UINT16    values[12];
528   } TPMS_COMMAND_AUDIT_INFO_mst;
529   typedef const struct TPMS_SESSION_AUDIT_INFO_mst
530   {
531       UINT8     marshalType;
532       UINT8     elements;
533       UINT16    values[6];
534   } TPMS_SESSION_AUDIT_INFO_mst;
535   typedef const struct TPMS_CREATION_INFO_mst
536   {
537       UINT8     marshalType;
538       UINT8     elements;
539       UINT16    values[6];
540   } TPMS_CREATION_INFO_mst;
541   typedef const struct TPMS_NV_CERTIFY_INFO_mst
542   {
543       UINT8     marshalType;
544       UINT8     elements;
545       UINT16    values[9];
546   } TPMS_NV_CERTIFY_INFO_mst;
547   typedef const struct TPMS_NV_DIGEST_CERTIFY_INFO_mst
548   {
549       UINT8     marshalType;
550       UINT8     elements;
551       UINT16    values[6];
552   } TPMS_NV_DIGEST_CERTIFY_INFO_mst;
553   typedef const struct TPMI_ST_ATTEST_mst {
554       UINT8         marshalType;
555       UINT8         modifiers;
556       UINT8         errorCode;
557       UINT8         ranges;
558       UINT8         singles;
559       UINT32        values[3];
560   } TPMI_ST_ATTEST_mst;
561   typedef struct TPMU_ATTEST_mst
562   {
563       BYTE            countOfselectors;
564       BYTE            modifiers;
565       UINT16        offsetOfUnmarshalTypes;
566       UINT32        selectors[8];
567       UINT16        marshalingTypes[8];
568   } TPMU_ATTEST_mst;
569   typedef const struct TPMS_ATTEST_mst
570   {
571       UINT8     marshalType;
572       UINT8     elements;
573       UINT16    values[21];
574   } TPMS_ATTEST_mst;
575   typedef const struct TPMS_AUTH_COMMAND_mst
576   {
577       UINT8     marshalType;
578       UINT8     elements;
579       UINT16    values[12];
580   } TPMS_AUTH_COMMAND_mst;
581   typedef const struct TPMS_AUTH_RESPONSE_mst
582   {
583       UINT8     marshalType;
```

```
584         UINT8     elements;
585         UINT16    values[9];
586     } TPMS_AUTH_RESPONSE_mst;
587     typedef const struct TPMI_TDES_KEY_BITS_mst {
588         UINT8         marshalType;
589         UINT8         modifiers;
590         UINT8         errorCode;
591         UINT8         entries;
592         UINT32        values[1];
593     } TPMI_TDES_KEY_BITS_mst;
594     typedef const struct TPMI_AES_KEY_BITS_mst {
595         UINT8         marshalType;
596         UINT8         modifiers;
597         UINT8         errorCode;
598         UINT8         entries;
599         UINT32        values[3];
600     } TPMI_AES_KEY_BITS_mst;
601     typedef const struct TPMI_SM4_KEY_BITS_mst {
602         UINT8         marshalType;
603         UINT8         modifiers;
604         UINT8         errorCode;
605         UINT8         entries;
606         UINT32        values[1];
607     } TPMI_SM4_KEY_BITS_mst;
608     typedef const struct TPMI_CAMELLIA_KEY_BITS_mst {
609         UINT8         marshalType;
610         UINT8         modifiers;
611         UINT8         errorCode;
612         UINT8         entries;
613         UINT32        values[3];
614     } TPMI_CAMELLIA_KEY_BITS_mst;
615     typedef struct TPMU_SYM_KEY_BITS_mst
616     {
617         BYTE            countOfselectors;
618         BYTE            modifiers;
619         UINT16        offsetOfUnmarshalTypes;
620         UINT32        selectors[6];
621         UINT16        marshalingTypes[6];
622     } TPMU_SYM_KEY_BITS_mst;
623     typedef struct TPMU_SYM_MODE_mst
624     {
625         BYTE            countOfselectors;
626         BYTE            modifiers;
627         UINT16        offsetOfUnmarshalTypes;
628         UINT32        selectors[6];
629         UINT16        marshalingTypes[6];
630     } TPMU_SYM_MODE_mst;
631     typedef const struct TPMT_SYM_DEF_mst
632     {
633         UINT8     marshalType;
634         UINT8     elements;
635         UINT16    values[9];
636     } TPMT_SYM_DEF_mst;
637     typedef const struct TPMT_SYM_DEF_OBJECT_mst
638     {
639         UINT8     marshalType;
640         UINT8     elements;
641         UINT16    values[9];
642     } TPMT_SYM_DEF_OBJECT_mst;
643     typedef const struct TPMS_SYMCIPHER_PARMS_mst
644     {
645         UINT8     marshalType;
646         UINT8     elements;
647         UINT16    values[3];
648     } TPMS_SYMCIPHER_PARMS_mst;
649     typedef const struct TPMS_DERIVE_mst
```

```
650 {
651     UINT8     marshalType;
652     UINT8     elements;
653     UINT16    values[6];
654 } TPMS_DERIVE_mst;
655 typedef const struct TPMS_SENSITIVE_CREATE_mst
656 {
657     UINT8     marshalType;
658     UINT8     elements;
659     UINT16    values[6];
660 } TPMS_SENSITIVE_CREATE_mst;
661 typedef const struct TPMS_SCHEME_HASH_mst
662 {
663     UINT8     marshalType;
664     UINT8     elements;
665     UINT16    values[3];
666 } TPMS_SCHEME_HASH_mst;
667 typedef const struct TPMS_SCHEME_ECDAA_mst
668 {
669     UINT8     marshalType;
670     UINT8     elements;
671     UINT16    values[6];
672 } TPMS_SCHEME_ECDAA_mst;
673 typedef const struct TPMI_ALG_KEYEDHASH_SCHEME_mst {
674     UINT8       marshalType;
675     UINT8       modifiers;
676     UINT8       errorCode;
677     UINT32      values[4];
678 } TPMI_ALG_KEYEDHASH_SCHEME_mst;
679 typedef const struct TPMS_SCHEME_XOR_mst
680 {
681     UINT8     marshalType;
682     UINT8     elements;
683     UINT16    values[6];
684 } TPMS_SCHEME_XOR_mst;
685 typedef struct TPMU_SCHEME_KEYEDHASH_mst
686 {
687     BYTE            countOfselectors;
688     BYTE            modifiers;
689     UINT16        offsetOfUnmarshalTypes;
690     UINT32        selectors[3];
691     UINT16        marshalingTypes[3];
692 } TPMU_SCHEME_KEYEDHASH_mst;
693 typedef const struct TPMT_KEYEDHASH_SCHEME_mst
694 {
695     UINT8     marshalType;
696     UINT8     elements;
697     UINT16    values[6];
698 } TPMT_KEYEDHASH_SCHEME_mst;
699 typedef struct TPMU_SIG_SCHEME_mst
700 {
701     BYTE            countOfselectors;
702     BYTE            modifiers;
703     UINT16        offsetOfUnmarshalTypes;
704     UINT32        selectors[8];
705     UINT16        marshalingTypes[8];
706 } TPMU_SIG_SCHEME_mst;
707 typedef const struct TPMT_SIG_SCHEME_mst
708 {
709     UINT8     marshalType;
710     UINT8     elements;
711     UINT16    values[6];
712 } TPMT_SIG_SCHEME_mst;
713 typedef struct TPMU_KDF_SCHEME_mst
714 {
715     BYTE            countOfselectors;
```

```
716         BYTE              modifiers;
717         UINT16          offsetOfUnmarshalTypes;
718         UINT32          selectors[5];
719         UINT16          marshalingTypes[5];
720     } TPMU_KDF_SCHEME_mst;
721     typedef const struct TPMT_KDF_SCHEME_mst
722     {
723         UINT8       marshalType;
724         UINT8       elements;
725         UINT16      values[6];
726     } TPMT_KDF_SCHEME_mst;
727     typedef const struct TPMI_ALG_ASYM_SCHEME_mst {
728         UINT8         marshalType;
729         UINT8         modifiers;
730         UINT8         errorCode;
731         UINT32        values[4];
732     } TPMI_ALG_ASYM_SCHEME_mst;
733     typedef struct TPMU_ASYM_SCHEME_mst
734     {
735         BYTE              countOfselectors;
736         BYTE              modifiers;
737         UINT16          offsetOfUnmarshalTypes;
738         UINT32          selectors[11];
739         UINT16          marshalingTypes[11];
740     } TPMU_ASYM_SCHEME_mst;
741     typedef const struct TPMI_ALG_RSA_SCHEME_mst {
742         UINT8         marshalType;
743         UINT8         modifiers;
744         UINT8         errorCode;
745         UINT32        values[4];
746     } TPMI_ALG_RSA_SCHEME_mst;
747     typedef const struct TPMT_RSA_SCHEME_mst
748     {
749         UINT8       marshalType;
750         UINT8       elements;
751         UINT16      values[6];
752     } TPMT_RSA_SCHEME_mst;
753     typedef const struct TPMI_ALG_RSA_DECRYPT_mst {
754         UINT8         marshalType;
755         UINT8         modifiers;
756         UINT8         errorCode;
757         UINT32        values[4];
758     } TPMI_ALG_RSA_DECRYPT_mst;
759     typedef const struct TPMT_RSA_DECRYPT_mst
760     {
761         UINT8       marshalType;
762         UINT8       elements;
763         UINT16      values[6];
764     } TPMT_RSA_DECRYPT_mst;
765     typedef const struct TPMI_RSA_KEY_BITS_mst {
766         UINT8           marshalType;
767         UINT8           modifiers;
768         UINT8           errorCode;
769         UINT8           entries;
770         UINT32          values[3];
771     } TPMI_RSA_KEY_BITS_mst;
772     typedef const struct TPMS_ECC_POINT_mst
773     {
774         UINT8       marshalType;
775         UINT8       elements;
776         UINT16      values[6];
777     } TPMS_ECC_POINT_mst;
778     typedef const struct TPMI_ALG_ECC_SCHEME_mst {
779         UINT8         marshalType;
780         UINT8         modifiers;
781         UINT8         errorCode;
```

```
782         UINT32      values[4];
783     } TPMI_ALG_ECC_SCHEME_mst;
784     typedef const struct TPMI_ECC_CURVE_mst {
785         UINT8       marshalType;
786         UINT8       modifiers;
787         UINT8       errorCode;
788         UINT32      values[3];
789     } TPMI_ECC_CURVE_mst;
790     typedef const struct TPMT_ECC_SCHEME_mst
791     {
792         UINT8     marshalType;
793         UINT8     elements;
794         UINT16    values[6];
795     } TPMT_ECC_SCHEME_mst;
796     typedef const struct TPMS_ALGORITHM_DETAIL_ECC_mst
797     {
798         UINT8     marshalType;
799         UINT8     elements;
800         UINT16    values[33];
801     } TPMS_ALGORITHM_DETAIL_ECC_mst;
802     typedef const struct TPMS_SIGNATURE_RSA_mst
803     {
804         UINT8     marshalType;
805         UINT8     elements;
806         UINT16    values[6];
807     } TPMS_SIGNATURE_RSA_mst;
808     typedef const struct TPMS_SIGNATURE_ECC_mst
809     {
810         UINT8     marshalType;
811         UINT8     elements;
812         UINT16    values[9];
813     } TPMS_SIGNATURE_ECC_mst;
814     typedef struct TPMU_SIGNATURE_mst
815     {
816         BYTE            countOfselectors;
817         BYTE            modifiers;
818         UINT16        offsetOfUnmarshalTypes;
819         UINT32        selectors[8];
820         UINT16        marshalingTypes[8];
821     } TPMU_SIGNATURE_mst;
822     typedef const struct TPMT_SIGNATURE_mst
823     {
824         UINT8     marshalType;
825         UINT8     elements;
826         UINT16    values[6];
827     } TPMT_SIGNATURE_mst;
828     typedef struct TPMU_ENCRYPTED_SECRET_mst
829     {
830         BYTE            countOfselectors;
831         BYTE            modifiers;
832         UINT16        offsetOfUnmarshalTypes;
833         UINT32        selectors[4];
834         UINT16        marshalingTypes[4];
835     } TPMU_ENCRYPTED_SECRET_mst;
836     typedef const struct TPMI_ALG_PUBLIC_mst {
837         UINT8       marshalType;
838         UINT8       modifiers;
839         UINT8       errorCode;
840         UINT32      values[4];
841     } TPMI_ALG_PUBLIC_mst;
842     typedef struct TPMU_PUBLIC_ID_mst
843     {
844         BYTE            countOfselectors;
845         BYTE            modifiers;
846         UINT16        offsetOfUnmarshalTypes;
847         UINT32        selectors[4];
```

```
848         UINT16        marshalingTypes[4];
849     } TPMU_PUBLIC_ID_mst;
850     typedef const struct TPMS_KEYEDHASH_PARMS_mst
851     {
852         UINT8     marshalType;
853         UINT8     elements;
854         UINT16    values[3];
855     } TPMS_KEYEDHASH_PARMS_mst;
856     typedef const struct TPMS_RSA_PARMS_mst
857     {
858         UINT8     marshalType;
859         UINT8     elements;
860         UINT16    values[12];
861     } TPMS_RSA_PARMS_mst;
862     typedef const struct TPMS_ECC_PARMS_mst
863     {
864         UINT8     marshalType;
865         UINT8     elements;
866         UINT16    values[12];
867     } TPMS_ECC_PARMS_mst;
868     typedef struct TPMU_PUBLIC_PARMS_mst
869     {
870         BYTE            countOfselectors;
871         BYTE            modifiers;
872         UINT16        offsetOfUnmarshalTypes;
873         UINT32        selectors[4];
874         UINT16        marshalingTypes[4];
875     } TPMU_PUBLIC_PARMS_mst;
876     typedef const struct TPMT_PUBLIC_PARMS_mst
877     {
878         UINT8     marshalType;
879         UINT8     elements;
880         UINT16    values[6];
881     } TPMT_PUBLIC_PARMS_mst;
882     typedef const struct TPMT_PUBLIC_mst
883     {
884         UINT8     marshalType;
885         UINT8     elements;
886         UINT16    values[18];
887     } TPMT_PUBLIC_mst;
888     typedef struct TPMU_SENSITIVE_COMPOSITE_mst
889     {
890         BYTE            countOfselectors;
891         BYTE            modifiers;
892         UINT16        offsetOfUnmarshalTypes;
893         UINT32        selectors[4];
894         UINT16        marshalingTypes[4];
895     } TPMU_SENSITIVE_COMPOSITE_mst;
896     typedef const struct TPMT_SENSITIVE_mst
897     {
898         UINT8     marshalType;
899         UINT8     elements;
900         UINT16    values[12];
901     } TPMT_SENSITIVE_mst;
902     typedef const struct TPMS_NV_PIN_COUNTER_PARAMETERS_mst
903     {
904         UINT8     marshalType;
905         UINT8     elements;
906         UINT16    values[6];
907     } TPMS_NV_PIN_COUNTER_PARAMETERS_mst;
908     typedef const struct TPMS_NV_PUBLIC_mst
909     {
910         UINT8     marshalType;
911         UINT8     elements;
912         UINT16    values[15];
913     } TPMS_NV_PUBLIC_mst;
```

```
914   typedef const struct TPMS_CONTEXT_DATA_mst
915   {
916       UINT8     marshalType;
917       UINT8     elements;
918       UINT16    values[6];
919   } TPMS_CONTEXT_DATA_mst;
920   typedef const struct TPMS_CONTEXT_mst
921   {
922       UINT8     marshalType;
923       UINT8     elements;
924       UINT16    values[12];
925   } TPMS_CONTEXT_mst;
926   typedef const struct TPMS_CREATION_DATA_mst
927   {
928       UINT8     marshalType;
929       UINT8     elements;
930       UINT16    values[21];
931   } TPMS_CREATION_DATA_mst;
932   typedef const struct TPM_AT_mst {
933       UINT8         marshalType;
934       UINT8         modifiers;
935       UINT8         errorCode;
936       UINT8         entries;
937       UINT32        values[4];
938   } TPM_AT_mst;
939   typedef const struct TPMS_AC_OUTPUT_mst
940   {
941       UINT8     marshalType;
942       UINT8     elements;
943       UINT16    values[6];
944   } TPMS_AC_OUTPUT_mst;
945   typedef const struct Type02_mst {
946       UINT8       marshalType;
947       UINT8       modifiers;
948       UINT8       errorCode;
949       UINT32      values[2];
950   } Type02_mst;
951   typedef const struct Type03_mst {
952       UINT8       marshalType;
953       UINT8       modifiers;
954       UINT8       errorCode;
955       UINT32      values[2];
956   } Type03_mst;
957   typedef const struct Type04_mst {
958       UINT8       marshalType;
959       UINT8       modifiers;
960       UINT8       errorCode;
961       UINT32      values[2];
962   } Type04_mst;
963   typedef const struct Type06_mst {
964       UINT8       marshalType;
965       UINT8       modifiers;
966       UINT8       errorCode;
967       UINT32      values[2];
968   } Type06_mst;
969   typedef const struct Type08_mst {
970       UINT8       marshalType;
971       UINT8       modifiers;
972       UINT8       errorCode;
973       UINT32      values[2];
974   } Type08_mst;
975   typedef const struct Type10_mst {
976       UINT8         marshalType;
977       UINT8         modifiers;
978       UINT8         errorCode;
979       UINT8         entries;
```

```
980         UINT32          values[1];
981     } Type10_mst;
982     typedef const struct Type11_mst {
983         UINT8           marshalType;
984         UINT8           modifiers;
985         UINT8           errorCode;
986         UINT8           entries;
987         UINT32          values[1];
988     } Type11_mst;
989     typedef const struct Type12_mst {
990         UINT8           marshalType;
991         UINT8           modifiers;
992         UINT8           errorCode;
993         UINT8           entries;
994         UINT32          values[2];
995     } Type12_mst;
996     typedef const struct Type13_mst {
997         UINT8           marshalType;
998         UINT8           modifiers;
999         UINT8           errorCode;
1000        UINT8           entries;
1001        UINT32          values[1];
1002    } Type13_mst;
1003    typedef const struct Type15_mst {
1004        UINT8         marshalType;
1005        UINT8         modifiers;
1006        UINT8         errorCode;
1007        UINT32        values[2];
1008    } Type15_mst;
1009    typedef const struct Type17_mst {
1010        UINT8         marshalType;
1011        UINT8         modifiers;
1012        UINT8         errorCode;
1013        UINT32        values[2];
1014    } Type17_mst;
1015    typedef const struct Type18_mst {
1016        UINT8         marshalType;
1017        UINT8         modifiers;
1018        UINT8         errorCode;
1019        UINT32        values[2];
1020    } Type18_mst;
1021    typedef const struct Type19_mst {
1022        UINT8         marshalType;
1023        UINT8         modifiers;
1024        UINT8         errorCode;
1025        UINT32        values[2];
1026    } Type19_mst;
1027    typedef const struct Type20_mst {
1028        UINT8         marshalType;
1029        UINT8         modifiers;
1030        UINT8         errorCode;
1031        UINT32        values[2];
1032    } Type20_mst;
1033    typedef const struct Type22_mst {
1034        UINT8         marshalType;
1035        UINT8         modifiers;
1036        UINT8         errorCode;
1037        UINT32        values[2];
1038    } Type22_mst;
1039    typedef const struct Type23_mst {
1040        UINT8         marshalType;
1041        UINT8         modifiers;
1042        UINT8         errorCode;
1043        UINT32        values[2];
1044    } Type23_mst;
1045    typedef const struct Type24_mst {
```

```
1046        UINT8       marshalType;
1047        UINT8       modifiers;
1048        UINT8       errorCode;
1049        UINT32      values[2];
1050    } Type24_mst;
1051    typedef const struct Type25_mst {
1052        UINT8       marshalType;
1053        UINT8       modifiers;
1054        UINT8       errorCode;
1055        UINT32      values[2];
1056    } Type25_mst;
1057    typedef const struct Type26_mst {
1058        UINT8       marshalType;
1059        UINT8       modifiers;
1060        UINT8       errorCode;
1061        UINT32      values[2];
1062    } Type26_mst;
1063    typedef const struct Type27_mst {
1064        UINT8       marshalType;
1065        UINT8       modifiers;
1066        UINT8       errorCode;
1067        UINT32      values[2];
1068    } Type27_mst;
1069    typedef const struct Type29_mst {
1070        UINT8       marshalType;
1071        UINT8       modifiers;
1072        UINT8       errorCode;
1073        UINT32      values[2];
1074    } Type29_mst;
1075    typedef const struct Type30_mst {
1076        UINT8       marshalType;
1077        UINT8       modifiers;
1078        UINT8       errorCode;
1079        UINT32      values[2];
1080    } Type30_mst;
1081    typedef const struct Type33_mst {
1082        UINT8       marshalType;
1083        UINT8       modifiers;
1084        UINT8       errorCode;
1085        UINT32      values[2];
1086    } Type33_mst;
1087    typedef const struct Type34_mst {
1088        UINT8       marshalType;
1089        UINT8       modifiers;
1090        UINT8       errorCode;
1091        UINT32      values[2];
1092    } Type34_mst;
1093    typedef const struct Type35_mst {
1094        UINT8       marshalType;
1095        UINT8       modifiers;
1096        UINT8       errorCode;
1097        UINT32      values[2];
1098    } Type35_mst;
1099    typedef const struct Type38_mst {
1100        UINT8       marshalType;
1101        UINT8       modifiers;
1102        UINT8       errorCode;
1103        UINT32      values[2];
1104    } Type38_mst;
1105    typedef const struct Type41_mst {
1106        UINT8       marshalType;
1107        UINT8       modifiers;
1108        UINT8       errorCode;
1109        UINT32      values[2];
1110    } Type41_mst;
1111    typedef const struct Type42_mst {
```

```
1112        UINT8       marshalType;
1113        UINT8       modifiers;
1114        UINT8       errorCode;
1115        UINT32      values[2];
1116    } Type42_mst;
1117    typedef const struct Type44_mst {
1118        UINT8       marshalType;
1119        UINT8       modifiers;
1120        UINT8       errorCode;
1121        UINT32      values[2];
1122    } Type44_mst;
```

This structure combines all the individual marshaling structures to build something that can be referenced by offset rather than full address

```
1123    typedef const struct MarshalData_st {
1124        UintMarshal_mst                 UINT8_DATA;
1125        UintMarshal_mst                 UINT16_DATA;
1126        UintMarshal_mst                 UINT32_DATA;
1127        UintMarshal_mst                 UINT64_DATA;
1128        UintMarshal_mst                 INT8_DATA;
1129        UintMarshal_mst                 INT16_DATA;
1130        UintMarshal_mst                 INT32_DATA;
1131        UintMarshal_mst                 INT64_DATA;
1132        UintMarshal_mst                 UINT0_DATA;
1133        TPM_ECC_CURVE_mst               TPM_ECC_CURVE_DATA;
1134        TPM_CLOCK_ADJUST_mst            TPM_CLOCK_ADJUST_DATA;
1135        TPM_EO_mst                      TPM_EO_DATA;
1136        TPM_SU_mst                      TPM_SU_DATA;
1137        TPM_SE_mst                      TPM_SE_DATA;
1138        TPM_CAP_mst                     TPM_CAP_DATA;
1139        AttributesMarshal_mst           TPMA_ALGORITHM_DATA;
1140        AttributesMarshal_mst           TPMA_OBJECT_DATA;
1141        AttributesMarshal_mst           TPMA_SESSION_DATA;
1142        AttributesMarshal_mst           TPMA_ACT_DATA;
1143        TPMI_YES_NO_mst                 TPMI_YES_NO_DATA;
1144        TPMI_DH_OBJECT_mst              TPMI_DH_OBJECT_DATA;
1145        TPMI_DH_PARENT_mst              TPMI_DH_PARENT_DATA;
1146        TPMI_DH_PERSISTENT_mst          TPMI_DH_PERSISTENT_DATA;
1147        TPMI_DH_ENTITY_mst              TPMI_DH_ENTITY_DATA;
1148        TPMI_DH_PCR_mst                 TPMI_DH_PCR_DATA;
1149        TPMI_SH_AUTH_SESSION_mst        TPMI_SH_AUTH_SESSION_DATA;
1150        TPMI_SH_HMAC_mst                TPMI_SH_HMAC_DATA;
1151        TPMI_SH_POLICY_mst              TPMI_SH_POLICY_DATA;
1152        TPMI_DH_CONTEXT_mst             TPMI_DH_CONTEXT_DATA;
1153        TPMI_DH_SAVED_mst               TPMI_DH_SAVED_DATA;
1154        TPMI_RH_HIERARCHY_mst           TPMI_RH_HIERARCHY_DATA;
1155        TPMI_RH_ENABLES_mst             TPMI_RH_ENABLES_DATA;
1156        TPMI_RH_HIERARCHY_AUTH_mst      TPMI_RH_HIERARCHY_AUTH_DATA;
1157        TPMI_RH_HIERARCHY_POLICY_mst    TPMI_RH_HIERARCHY_POLICY_DATA;
1158        TPMI_RH_PLATFORM_mst            TPMI_RH_PLATFORM_DATA;
1159        TPMI_RH_OWNER_mst               TPMI_RH_OWNER_DATA;
1160        TPMI_RH_ENDORSEMENT_mst         TPMI_RH_ENDORSEMENT_DATA;
1161        TPMI_RH_PROVISION_mst           TPMI_RH_PROVISION_DATA;
1162        TPMI_RH_CLEAR_mst               TPMI_RH_CLEAR_DATA;
1163        TPMI_RH_NV_AUTH_mst             TPMI_RH_NV_AUTH_DATA;
1164        TPMI_RH_LOCKOUT_mst             TPMI_RH_LOCKOUT_DATA;
1165        TPMI_RH_NV_INDEX_mst            TPMI_RH_NV_INDEX_DATA;
1166        TPMI_RH_AC_mst                  TPMI_RH_AC_DATA;
1167        TPMI_RH_ACT_mst                 TPMI_RH_ACT_DATA;
1168        TPMI_ALG_HASH_mst               TPMI_ALG_HASH_DATA;
1169        TPMI_ALG_ASYM_mst               TPMI_ALG_ASYM_DATA;
1170        TPMI_ALG_SYM_mst                TPMI_ALG_SYM_DATA;
1171        TPMI_ALG_SYM_OBJECT_mst         TPMI_ALG_SYM_OBJECT_DATA;
1172        TPMI_ALG_SYM_MODE_mst           TPMI_ALG_SYM_MODE_DATA;
```

```
1173        TPMI_ALG_KDF_mst                TPMI_ALG_KDF_DATA;
1174        TPMI_ALG_SIG_SCHEME_mst         TPMI_ALG_SIG_SCHEME_DATA;
1175        TPMI_ECC_KEY_EXCHANGE_mst       TPMI_ECC_KEY_EXCHANGE_DATA;
1176        TPMI_ST_COMMAND_TAG_mst         TPMI_ST_COMMAND_TAG_DATA;
1177        TPMI_ALG_MAC_SCHEME_mst         TPMI_ALG_MAC_SCHEME_DATA;
1178        TPMI_ALG_CIPHER_MODE_mst        TPMI_ALG_CIPHER_MODE_DATA;
1179        TPMS_EMPTY_mst                  TPMS_EMPTY_DATA;
1180        TPMS_ALGORITHM_DESCRIPTION_mst  TPMS_ALGORITHM_DESCRIPTION_DATA;
1181        TPMU_HA_mst                     TPMU_HA_DATA;
1182        TPMT_HA_mst                     TPMT_HA_DATA;
1183        Tpm2bMarshal_mst                TPM2B_DIGEST_DATA;
1184        Tpm2bMarshal_mst                TPM2B_DATA_DATA;
1185        Tpm2bMarshal_mst                TPM2B_EVENT_DATA;
1186        Tpm2bMarshal_mst                TPM2B_MAX_BUFFER_DATA;
1187        Tpm2bMarshal_mst                TPM2B_MAX_NV_BUFFER_DATA;
1188        Tpm2bMarshal_mst                TPM2B_TIMEOUT_DATA;
1189        Tpm2bMarshal_mst                TPM2B_IV_DATA;
1190        NullUnionMarshal_mst            NULL_UNION_DATA;
1191        Tpm2bMarshal_mst                TPM2B_NAME_DATA;
1192        TPMS_PCR_SELECT_mst             TPMS_PCR_SELECT_DATA;
1193        TPMS_PCR_SELECTION_mst          TPMS_PCR_SELECTION_DATA;
1194        TPMT_TK_CREATION_mst            TPMT_TK_CREATION_DATA;
1195        TPMT_TK_VERIFIED_mst            TPMT_TK_VERIFIED_DATA;
1196        TPMT_TK_AUTH_mst                TPMT_TK_AUTH_DATA;
1197        TPMT_TK_HASHCHECK_mst           TPMT_TK_HASHCHECK_DATA;
1198        TPMS_ALG_PROPERTY_mst           TPMS_ALG_PROPERTY_DATA;
1199        TPMS_TAGGED_PROPERTY_mst        TPMS_TAGGED_PROPERTY_DATA;
1200        TPMS_TAGGED_PCR_SELECT_mst      TPMS_TAGGED_PCR_SELECT_DATA;
1201        TPMS_TAGGED_POLICY_mst          TPMS_TAGGED_POLICY_DATA;
1202        TPMS_ACT_DATA_mst               TPMS_ACT_DATA_DATA;
1203        ListMarshal_mst                 TPML_CC_DATA;
1204        ListMarshal_mst                 TPML_CCA_DATA;
1205        ListMarshal_mst                 TPML_ALG_DATA;
1206        ListMarshal_mst                 TPML_HANDLE_DATA;
1207        ListMarshal_mst                 TPML_DIGEST_DATA;
1208        ListMarshal_mst                 TPML_DIGEST_VALUES_DATA;
1209        ListMarshal_mst                 TPML_PCR_SELECTION_DATA;
1210        ListMarshal_mst                 TPML_ALG_PROPERTY_DATA;
1211        ListMarshal_mst                 TPML_TAGGED_TPM_PROPERTY_DATA;
1212        ListMarshal_mst                 TPML_TAGGED_PCR_PROPERTY_DATA;
1213        ListMarshal_mst                 TPML_ECC_CURVE_DATA;
1214        ListMarshal_mst                 TPML_TAGGED_POLICY_DATA;
1215        ListMarshal_mst                 TPML_ACT_DATA_DATA;
1216        TPMU_CAPABILITIES_mst           TPMU_CAPABILITIES_DATA;
1217        TPMS_CAPABILITY_DATA_mst        TPMS_CAPABILITY_DATA_DATA;
1218        TPMS_CLOCK_INFO_mst             TPMS_CLOCK_INFO_DATA;
1219        TPMS_TIME_INFO_mst              TPMS_TIME_INFO_DATA;
1220        TPMS_TIME_ATTEST_INFO_mst       TPMS_TIME_ATTEST_INFO_DATA;
1221        TPMS_CERTIFY_INFO_mst           TPMS_CERTIFY_INFO_DATA;
1222        TPMS_QUOTE_INFO_mst             TPMS_QUOTE_INFO_DATA;
1223        TPMS_COMMAND_AUDIT_INFO_mst     TPMS_COMMAND_AUDIT_INFO_DATA;
1224        TPMS_SESSION_AUDIT_INFO_mst     TPMS_SESSION_AUDIT_INFO_DATA;
1225        TPMS_CREATION_INFO_mst          TPMS_CREATION_INFO_DATA;
1226        TPMS_NV_CERTIFY_INFO_mst        TPMS_NV_CERTIFY_INFO_DATA;
1227        TPMS_NV_DIGEST_CERTIFY_INFO_mst TPMS_NV_DIGEST_CERTIFY_INFO_DATA;
1228        TPMI_ST_ATTEST_mst              TPMI_ST_ATTEST_DATA;
1229        TPMU_ATTEST_mst                 TPMU_ATTEST_DATA;
1230        TPMS_ATTEST_mst                 TPMS_ATTEST_DATA;
1231        Tpm2bMarshal_mst                TPM2B_ATTEST_DATA;
1232        TPMS_AUTH_COMMAND_mst           TPMS_AUTH_COMMAND_DATA;
1233        TPMS_AUTH_RESPONSE_mst          TPMS_AUTH_RESPONSE_DATA;
1234        TPMI_TDES_KEY_BITS_mst          TPMI_TDES_KEY_BITS_DATA;
1235        TPMI_AES_KEY_BITS_mst           TPMI_AES_KEY_BITS_DATA;
1236        TPMI_SM4_KEY_BITS_mst           TPMI_SM4_KEY_BITS_DATA;
1237        TPMI_CAMELLIA_KEY_BITS_mst      TPMI_CAMELLIA_KEY_BITS_DATA;
1238        TPMU_SYM_KEY_BITS_mst           TPMU_SYM_KEY_BITS_DATA;
```

```
1239        TPMU_SYM_MODE_mst               TPMU_SYM_MODE_DATA;
1240        TPMT_SYM_DEF_mst                TPMT_SYM_DEF_DATA;
1241        TPMT_SYM_DEF_OBJECT_mst         TPMT_SYM_DEF_OBJECT_DATA;
1242        Tpm2bMarshal_mst                TPM2B_SYM_KEY_DATA;
1243        TPMS_SYMCIPHER_PARMS_mst        TPMS_SYMCIPHER_PARMS_DATA;
1244        Tpm2bMarshal_mst                TPM2B_LABEL_DATA;
1245        TPMS_DERIVE_mst                 TPMS_DERIVE_DATA;
1246        Tpm2bMarshal_mst                TPM2B_DERIVE_DATA;
1247        Tpm2bMarshal_mst                TPM2B_SENSITIVE_DATA_DATA;
1248        TPMS_SENSITIVE_CREATE_mst       TPMS_SENSITIVE_CREATE_DATA;
1249        Tpm2bsMarshal_mst               TPM2B_SENSITIVE_CREATE_DATA;
1250        TPMS_SCHEME_HASH_mst            TPMS_SCHEME_HASH_DATA;
1251        TPMS_SCHEME_ECDAA_mst           TPMS_SCHEME_ECDAA_DATA;
1252        TPMI_ALG_KEYEDHASH_SCHEME_mst   TPMI_ALG_KEYEDHASH_SCHEME_DATA;
1253        TPMS_SCHEME_XOR_mst             TPMS_SCHEME_XOR_DATA;
1254        TPMU_SCHEME_KEYEDHASH_mst       TPMU_SCHEME_KEYEDHASH_DATA;
1255        TPMT_KEYEDHASH_SCHEME_mst       TPMT_KEYEDHASH_SCHEME_DATA;
1256        TPMU_SIG_SCHEME_mst             TPMU_SIG_SCHEME_DATA;
1257        TPMT_SIG_SCHEME_mst             TPMT_SIG_SCHEME_DATA;
1258        TPMU_KDF_SCHEME_mst             TPMU_KDF_SCHEME_DATA;
1259        TPMT_KDF_SCHEME_mst             TPMT_KDF_SCHEME_DATA;
1260        TPMI_ALG_ASYM_SCHEME_mst        TPMI_ALG_ASYM_SCHEME_DATA;
1261        TPMU_ASYM_SCHEME_mst            TPMU_ASYM_SCHEME_DATA;
1262        TPMI_ALG_RSA_SCHEME_mst         TPMI_ALG_RSA_SCHEME_DATA;
1263        TPMT_RSA_SCHEME_mst             TPMT_RSA_SCHEME_DATA;
1264        TPMI_ALG_RSA_DECRYPT_mst        TPMI_ALG_RSA_DECRYPT_DATA;
1265        TPMT_RSA_DECRYPT_mst            TPMT_RSA_DECRYPT_DATA;
1266        Tpm2bMarshal_mst                TPM2B_PUBLIC_KEY_RSA_DATA;
1267        TPMI_RSA_KEY_BITS_mst           TPMI_RSA_KEY_BITS_DATA;
1268        Tpm2bMarshal_mst                TPM2B_PRIVATE_KEY_RSA_DATA;
1269        Tpm2bMarshal_mst                TPM2B_ECC_PARAMETER_DATA;
1270        TPMS_ECC_POINT_mst              TPMS_ECC_POINT_DATA;
1271        Tpm2bsMarshal_mst               TPM2B_ECC_POINT_DATA;
1272        TPMI_ALG_ECC_SCHEME_mst         TPMI_ALG_ECC_SCHEME_DATA;
1273        TPMI_ECC_CURVE_mst              TPMI_ECC_CURVE_DATA;
1274        TPMT_ECC_SCHEME_mst             TPMT_ECC_SCHEME_DATA;
1275        TPMS_ALGORITHM_DETAIL_ECC_mst   TPMS_ALGORITHM_DETAIL_ECC_DATA;
1276        TPMS_SIGNATURE_RSA_mst          TPMS_SIGNATURE_RSA_DATA;
1277        TPMS_SIGNATURE_ECC_mst          TPMS_SIGNATURE_ECC_DATA;
1278        TPMU_SIGNATURE_mst              TPMU_SIGNATURE_DATA;
1279        TPMT_SIGNATURE_mst              TPMT_SIGNATURE_DATA;
1280        TPMU_ENCRYPTED_SECRET_mst       TPMU_ENCRYPTED_SECRET_DATA;
1281        Tpm2bMarshal_mst                TPM2B_ENCRYPTED_SECRET_DATA;
1282        TPMI_ALG_PUBLIC_mst             TPMI_ALG_PUBLIC_DATA;
1283        TPMU_PUBLIC_ID_mst              TPMU_PUBLIC_ID_DATA;
1284        TPMS_KEYEDHASH_PARMS_mst        TPMS_KEYEDHASH_PARMS_DATA;
1285        TPMS_RSA_PARMS_mst              TPMS_RSA_PARMS_DATA;
1286        TPMS_ECC_PARMS_mst              TPMS_ECC_PARMS_DATA;
1287        TPMU_PUBLIC_PARMS_mst           TPMU_PUBLIC_PARMS_DATA;
1288        TPMT_PUBLIC_PARMS_mst           TPMT_PUBLIC_PARMS_DATA;
1289        TPMT_PUBLIC_mst                 TPMT_PUBLIC_DATA;
1290        Tpm2bsMarshal_mst               TPM2B_PUBLIC_DATA;
1291        Tpm2bMarshal_mst                TPM2B_TEMPLATE_DATA;
1292        Tpm2bMarshal_mst                TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA;
1293        TPMU_SENSITIVE_COMPOSITE_mst    TPMU_SENSITIVE_COMPOSITE_DATA;
1294        TPMT_SENSITIVE_mst              TPMT_SENSITIVE_DATA;
1295        Tpm2bsMarshal_mst               TPM2B_SENSITIVE_DATA;
1296        Tpm2bMarshal_mst                TPM2B_PRIVATE_DATA;
1297        Tpm2bMarshal_mst                TPM2B_ID_OBJECT_DATA;
1298        TPMS_NV_PIN_COUNTER_PARAMETERS_mst TPMS_NV_PIN_COUNTER_PARAMETERS_DATA;
1299        AttributesMarshal_mst           TPMA_NV_DATA;
1300        TPMS_NV_PUBLIC_mst              TPMS_NV_PUBLIC_DATA;
1301        Tpm2bsMarshal_mst               TPM2B_NV_PUBLIC_DATA;
1302        Tpm2bMarshal_mst                TPM2B_CONTEXT_SENSITIVE_DATA;
1303        TPMS_CONTEXT_DATA_mst           TPMS_CONTEXT_DATA_DATA;
1304        Tpm2bMarshal_mst                TPM2B_CONTEXT_DATA_DATA;
```

```
    TPMS_CONTEXT_mst                TPMS_CONTEXT_DATA;
    TPMS_CREATION_DATA_mst          TPMS_CREATION_DATA_DATA;
    Tpm2bsMarshal_mst               TPM2B_CREATION_DATA_DATA;
    TPM_AT_mst                      TPM_AT_DATA;
    TPMS_AC_OUTPUT_mst              TPMS_AC_OUTPUT_DATA;
    ListMarshal_mst                 TPML_AC_CAPABILITIES_DATA;
    MinMaxMarshal_mst               Type00_DATA;
    MinMaxMarshal_mst               Type01_DATA;
    Type02_mst                      Type02_DATA;
    Type03_mst                      Type03_DATA;
    Type04_mst                      Type04_DATA;
    MinMaxMarshal_mst               Type05_DATA;
    Type06_mst                      Type06_DATA;
    MinMaxMarshal_mst               Type07_DATA;
    Type08_mst                      Type08_DATA;
    Type10_mst                      Type10_DATA;
    Type11_mst                      Type11_DATA;
    Type12_mst                      Type12_DATA;
    Type13_mst                      Type13_DATA;
    Type15_mst                      Type15_DATA;
    Type17_mst                      Type17_DATA;
    Type18_mst                      Type18_DATA;
    Type19_mst                      Type19_DATA;
    Type20_mst                      Type20_DATA;
    Type22_mst                      Type22_DATA;
    Type23_mst                      Type23_DATA;
    Type24_mst                      Type24_DATA;
    Type25_mst                      Type25_DATA;
    Type26_mst                      Type26_DATA;
    Type27_mst                      Type27_DATA;
    MinMaxMarshal_mst               Type28_DATA;
    Type29_mst                      Type29_DATA;
    Type30_mst                      Type30_DATA;
    MinMaxMarshal_mst               Type31_DATA;
    MinMaxMarshal_mst               Type32_DATA;
    Type33_mst                      Type33_DATA;
    Type34_mst                      Type34_DATA;
    Type35_mst                      Type35_DATA;
    MinMaxMarshal_mst               Type36_DATA;
    MinMaxMarshal_mst               Type37_DATA;
    Type38_mst                      Type38_DATA;
    MinMaxMarshal_mst               Type39_DATA;
    MinMaxMarshal_mst               Type40_DATA;
    Type41_mst                      Type41_DATA;
    Type42_mst                      Type42_DATA;
    MinMaxMarshal_mst               Type43_DATA;
    Type44_mst                      Type44_DATA;
} MarshalData_st;
#endif // _TABLE_MARSHAL_TYPES_H_
```

9.10.8 Table Marshal Source

9.10.8.1 TableDrivenMarshal.c

```
#include <assert.h>
#include "Tpm.h"
#include "Marshal.h"
#include "TableMarshal.h"
#if TABLE_DRIVEN_MARSHAL
extern ArrayMarshal_mst ArrayLookupTable[];

extern UINT16 MarshalLookupTable[];

typedef struct { int a; } External_Structure_t;
```

```
11
12  extern struct Exernal_Structure_t MarshalData;
13
14  #define IS_SUCCESS(UNMARSHAL_FUNCTION)                                          \
15      (TPM_RC_SUCCESS == (result = (UNMARSHAL_FUNCTION)))
16  marshalIndex_t IntegerDispatch[] = {
17      UINT8_MARSHAL_REF, UINT16_MARSHAL_REF, UINT32_MARSHAL_REF, UINT64_MARSHAL_REF,
18      INT8_MARSHAL_REF,  INT16_MARSHAL_REF,  INT32_MARSHAL_REF,  INT64_MARSHAL_REF
19  };
20
21  #if 1
22  #define GetDescriptor(reference)                                                \
23      ((MarshalHeader_mst *)(((BYTE *)(&MarshalData)) + (reference & NULL_MASK)))
24  #else
25  static const MarshalHeader_mst *GetDescriptor(marshalIndex_t index)
26  {
27      const MarshalHeader_mst    *mst = MarshalLookupTable[index & NULL_MASK];
28      return mst;
29  }
30  #endif
31  #define GetUnionDescriptor(_index_)                                             \
32      ((UnionMarshal_mst *)GetDescriptor(_index_))
33  #define GetArrayDescriptor(_index_)                                             \
34      ((ArrayMarshal_mst *))ArrayLookupTable[_index_ & NULL_MASK])
35
36  //*** GetUnmarshaledInteger()
37  // Gets the unmarshaled value and normalizes it to a UIN32 for other
38  // processing (comparisons and such).
39  static UINT32 GetUnmarshaledInteger(
40      marshalIndex_t       type,
41      const void          *target
42  )
43  {
44      int     size = (type & SIZE_MASK);
45  //
46      if(size == FOUR_BYTES)
47          return *((UINT32 *)target);
48      if(type & IS_SIGNED)
49      {
50          if(size == TWO_BYTES)
51              return (UINT32)*((int16_t *)target);
52          return (UINT32)*((int8_t *)target);
53      }
54      if(size == TWO_BYTES)
55          return (UINT32)*((UINT16 *)target);
56      return (UINT32)*((UINT8 *)target);
57  }
58  static UINT32 GetSelector(
59      void                *structure,
60      const UINT16        *values,
61      UINT16               descriptor
62  )
63  {
64      uint                 sel = GET_ELEMENT_NUMBER(descriptor);
65      // Get the offset of the value in the unmarshaled structure
66      const UINT16        *entry = &values[(sel * 3)];
67  //
68      return GetUnmarshaledInteger(GET_ELEMENT_SIZE(entry[0]),
69                           ((UINT8 *)structure) + entry[2]);
70  }
71  static TPM_RC UnmarshalBytes(
72      UINT8               *target,        // IN/OUT: place to put the bytes
73      UINT8               **buffer,       // IN/OUT: source of the input data
74      INT32               *size,          // IN/OUT: remaining bytes in the input buffer
75      int                  count          // IN: number of bytes to get
```

```
76  )
77  {
78      if((*size -= count) >= 0)
79      {
80          memcpy(target, *buffer, count);
81          *buffer += count;
82          return TPM_RC_SUCCESS;
83      }
84      return TPM_RC_INSUFFICIENT;
85  }
```

9.10.8.1.1.1 MarshalBytes()

Marshal an array of bytes.

```
 86  static UINT16 MarshalBytes(
 87      UINT8               *source,
 88      UINT8               **buffer,
 89      INT32               *size,
 90      int32_t              count
 91  )
 92  {
 93      if(buffer != NULL)
 94      {
 95          if(size != NULL && (size -= count) < 0)
 96              return 0;
 97          memcpy(*buffer, source, count);
 98          *buffer += count;
 99      }
100      return (UINT16)count;
101  }
```

9.10.8.1.1.2 ArrayUnmarshal()

Unmarshal an array. The *index* is of the form: *type*_ARRAY_MARSHAL_INDEX.

```
102   static TPM_RC ArrayUnmarshal(
103       UINT16               index,          // IN: the type of the array
104       UINT8               *target,         // IN: target for the data
105       UINT8               **buffer,        // IN/OUT: place to get the data
106       INT32               *size,           // IN/OUT: remaining unmarshal data
107       UINT32               count           // IN: number of values of 'index' to
108                                            //      unmarshal
109   )
110   {
111       marshalIndex_t       which = ArrayLookupTable[index & NULL_MASK].type;
112       UINT16               stride = ArrayLookupTable[index & NULL_MASK].stride;
113       TPM_RC               result;
114   //
115       if(stride == 1) // A byte array
116           result = UnmarshalBytes(target, buffer, size, count);
117       else
118       {
119           which |= index & NULL_FLAG;
120           for(result = TPM_RC_SUCCESS; count > 0; target += stride, count--)
121               if(!IS_SUCCESS(Unmarshal(which, target, buffer, size)))
122                   break;
123       }
124       return result;
125   }
```

9.10.8.1.1.3 ArrayMarshal()

```
126 static UINT16 ArrayMarshal(
127     UINT16               index,          // IN: the type of the array
128     UINT8               *source,         // IN: source of the data
129     UINT8               **buffer,        // IN/OUT: place to put the data
130     INT32               *size,           // IN/OUT: amount of space for the data
131     UINT32               count           // IN: number of values of 'index' to marshal
132 )
133 {
134     marshalIndex_t      which = ArrayLookupTable[index & NULL_MASK].type;
135     UINT16              stride = ArrayLookupTable[index & NULL_MASK].stride;
136     UINT16              retVal;
137 //
138     if(stride == 1) // A byte array
139         return MarshalBytes(source, buffer, size, count);
140     which |= index & NULL_FLAG;
141     for(retVal = 0
142         ; count > 0
143         ; source += stride, count--)
144         retVal += Marshal(which, source, buffer, size);
145
146     return retVal;
147 }
```

9.10.8.1.1.4 UnmarshalUnion()

```
148 TPM_RC
149 UnmarshalUnion(
150     UINT16               typeIndex,          // IN: the thing to unmarshal
151     void                *target,             // IN: were the data goes to
152     UINT8               **buffer,            // IN/OUT: the data source buffer
153     INT32               *size,               // IN/OUT: the remaining size
154     UINT32               selector
155 )
156 {
157     int                  i;
158     UnionMarshal_mst    *ut = GetUnionDescriptor(typeIndex);
159     marshalIndex_t       selected;
160 //
161     for(i = 0; i < ut->countOfselectors; i++)
162     {
163         if(selector == ut->selectors[i])
164         {
165             UINT8           *offset = ((UINT8 *)ut) + ut->offsetOfUnmarshalTypes;
166             // Get the selected thing to unmarshal
167             selected = ((marshalIndex_t *)offset)[i];
168             if(ut->modifiers & IS_ARRAY_UNION)
169                 return UnmarshalBytes(target, buffer, size, selected);
170             else
171             {
172                 // Propagate NULL_FLAG if the null flag was
173                 // propagated to the structure containing the union
174                 selected |= (typeIndex & NULL_FLAG);
175                 return Unmarshal(selected, target, buffer, size);
176             }
177         }
178     }
179     // Didn't find the value.
180     return TPM_RC_SELECTOR;
181 }
```

9.10.8.1.1.5 MarshalUnion()

```
182 UINT16
183 MarshalUnion(
184     UINT16               typeIndex,          // IN: the thing to marshal
185     void                *source,             // IN: were the data comes from
186     UINT8               **buffer,            // IN/OUT: the data source buffer
187     INT32               *size,               // IN/OUT: the remaining size
188     UINT32               selector            // IN: the union selector
189 )
190 {
191     int                  i;
192     UnionMarshal_mst    *ut = GetUnionDescriptor(typeIndex);
193     marshalIndex_t       selected;
194 //
195     for(i = 0; i < ut->countOfselectors; i++)
196     {
197         if(selector == ut->selectors[i])
198         {
199             UINT8           *offset = ((UINT8 *)ut) + ut->offsetOfUnmarshalTypes;
200             // Get the selected thing to unmarshal
201             selected = ((marshalIndex_t *)offset)[i];
202             if(ut->modifiers & IS_ARRAY_UNION)
203                 return MarshalBytes(source, buffer, size, selected);
204             else
205                 return Marshal(selected, source, buffer, size);
206         }
207     }
208     if(size != NULL)
209         *size = -1;
210     return 0;
211 }
212 TPM_RC
213 UnmarshalInteger(
214     int                  iSize,              // IN: Number of bytes in the integer
215     void                *target,             // OUT: receives the integer
216     UINT8               **buffer,            // IN/OUT: source of the data
217     INT32               *size,               // IN/OUT: amount of data available
218     UINT32              *value               // OUT: optional copy of 'target'
219 )
220 {
221     // This is just to save typing
222 #define _MB_ (*buffer)
223     // The size is a power of two so convert to regular integer
224     int                 bytes = (1 << (iSize & SIZE_MASK));
225 //
226     // Check to see if there is enough data to fulfill the request
227     if((*size -= bytes) >= 0)
228     {
229         // The most comon size
230         if(bytes == 4)
231         {
232             *((UINT32 *)target) = (UINT32)((((((_MB_[0] << 8) | _MB_[1]) << 8)
233                                              | _MB_[2]) << 8) | _MB_[3]);
234             // If a copy is needed, copy it.
235             if(value != NULL)
236                 *value = *((UINT32 *)target);
237         }
238         else if(bytes == 2)
239         {
240             *((UINT16 *)target) = (UINT16)((_MB_[0] << 8) | _MB_[1]);
241             // If a copy is needed, copy with the appropriate sign extension
242             if(value != NULL)
243             {
244                 if(iSize & IS_SIGNED)
```

```
245                     *value = (UINT32)(*((INT16 *)target));
246                 else
247                     *value = (UINT32)(*((UINT16 *)target));
248             }
249         }
250         else if(bytes == 1)
251         {
252             *((UINT8*)target) = (UINT8)_MB_[0];
253             // If a copy is needed, copy with the appropriate sign extension
254             if(value != NULL)
255             {
256                 if(iSize & IS_SIGNED)
257                     *value = (UINT32)(*((INT8 *)target));
258                 else
259                     *value = (UINT32)(*((UINT8 *)target));
260             }
261         }
262         else
263         {
264             // There is no input type that is a 64-bit value other than a UINT64. So
265             // there is no reason to do anything other than unmarshal it.
266             *((UINT64 *)target) = BYTE_ARRAY_TO_UINT64(*buffer);
267         }
268         *buffer += bytes;
269         return TPM_RC_SUCCESS;
270 #undef _MB_
271     }
272     return TPM_RC_INSUFFICIENT;
273 }
```

9.10.8.1.1.6 Unmarshal()

This is the function that performs unmarshaling of different numbered types. Each TPM type has a number. The number is used to lookup the address of the data structure that describes how to unmarshal that data type.

```
274  TPM_RC
275  Unmarshal(
276      UINT16               typeIndex,          // IN: the thing to marshal
277      void                *target,             // IN: were the data goes from
278      UINT8               **buffer,            // IN/OUT: the data source buffer
279      INT32               *size                // IN/OUT: the remaining size
280  )
281  {
282      const MarshalHeader_mst     *sel;
283      TPM_RC                       result;
284  //
285  #define _target    ((UINT8 *)target)
286      sel = GetDescriptor(typeIndex);
287      switch(sel->marshalType)
288      {
289          case UINT_MTYPE:
290          {
291              // A simple signed or unsigned integer value.
292              return UnmarshalInteger(sel->modifiers, target,
293                                      buffer, size, NULL);
294          }
295          case VALUES_MTYPE:
296          {
297              // This is the general-purpose structure that can handle things like
298              // TPMI_DH_PARENT that has multiple ranges, multiple singles and a
299              // 'null' value. When things cover a large range with holes in the range
300              // they can be turned into multiple ranges. There is no option for a bit
301              // field.
302              // The structure is:
303              //  typedef const struct ValuesMarshal_mst
304              //  {
305              //      UINT8           marshalType;        // VALUES_MTYPE
306              //      UINT8           modifiers;
307              //      UINT8           errorCode;
308              //      UINT8           ranges;
309              //      UINT8           singles;
310              //      UINT32          values[1];
311              //  } ValuesMarshal_mst;
312              // Unmarshal the base type
313              UINT32                   val;
314              if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
315                                             buffer, size, &val)))
316              {
317                  ValuesMarshal_mst   *vmt = ((ValuesMarshal_mst *)sel);
318                  const UINT32        *check = vmt->values;
319              //
320                  // if the TAKES_NULL flag is set, then the first entry in the values
321                  // list is the NULL value. Iy is not included in the 'ranges' or
322                  // 'singles' count.
323                  if((vmt->modifiers & TAKES_NULL) && (val == *check++))
324                  {
325                      if((typeIndex & NULL_FLAG) == 0)
326                          result = (TPM_RC)(sel->errorCode);
327                  }
328                  // No NULL value or input is not the NULL value
329                  else
330                  {
331                      int              i;
```

```
                //
                    // Check all the min-max ranges.
                    for(i = vmt->ranges - 1; i >= 0; check = &check[2], i--)
                        if((UINT32)(val - check[0]) <= check[1])
                            break;
                    // if the input is in a selected range, then i >= 0
                    if(i < 0)
                    {
                        // Not in any range, so check sigles
                        for(i = vmt->singles - 1; i >= 0; i--)
                            if(val == check[i])
                                break;
                    }
                    // If input not in range and not in any single so return error
                    if(i < 0)
                        result = (TPM_RC)(sel->errorCode);
                }
            }
            break;
        }
        case TABLE_MTYPE:
        {
            // This is a table with or without bit checking. The input is checked
            // against each value in the table. If the value is in the table, and
            // a bits table is present, then the bit field is checked to see if the
            // indiated value is implemented. For example, if there is a table of
            // allowed RSA key sises and the 2nd entry matches, then the 2nd bit in
            // thed bit field is check to see if that allowwed size is implemented in
            // this TPM.
            //  typedef const struct TableMarshal_mst
            //  {
            //      UINT8           marshalType;        // TABLE_MTYPE
            //      UINT8           modifiers;
            //      UINT8           errorCode;
            //      UINT8           singles;
            //      UINT32          values[1];
            //  } TableMarshal_mst;

            UINT32                  val;
        //
            // Unmarshal the base type
            if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
                                           buffer, size, &val)))
            {
                TableMarshal_mst    *tmt = ((TableMarshal_mst *)sel);
                const UINT32        *check = tmt->values;
            //
                // If this type has a null value, then it is the first value in the
                // list of values. It does not count in the count of values
                if((tmt->modifiers & TAKES_NULL) && (val == *check++))
                {
                    if((typeIndex & NULL_FLAG) == 0)
                        result = (TPM_RC)(sel->errorCode);
                }
                else
                {
                    int               i;
                //
                    // Process the singles
                    for(i = tmt->singles - 1; i >= 0; i--)
                    {
                        // does the input value match the value in the table
                        if(val == check[i])
                        {
                            // If there is an associated bit table, make sure that the
corresponding
```

```
397                             // bit is SET
398                             if((HAS_BITS & tmt->modifiers)
399                                && (!IS_BIT_SET32(i, &(check[tmt->singles]))))
400                                 // if not SET, then this is a failure.
401                                 i = -1;
402                             break;
403                         }
404                     }
405                     // error if not found or bit not SET
406                     if(i < 0)
407                         result = (TPM_RC)(sel->errorCode);
408                 }
409             }
410             break;
411         }
412         case MIN_MAX_MTYPE:
413         {
414             // A MIN_MAX is a range. It can have a bit field and a NULL value that is
415             // outside of the range. If the input value is in the min-max range then
416             // it is valid unless there is an associated bit field. Otherwise, it
417             // it is only valid if the corresponding value in the bit field is SET.
418             // The min value is 'values[0]' or 'values[1]' if there is a NULL value.
419             // The max value is the value after min. The max value is in the table as
420             // max minus min. This allows 'val' to be subtracted from min and then
421             // checked against max with one unsigned comparison. If present, the bit
422             // field will be the first 'values' after max.
423             //  typedef const struct MinMaxMarshal_mst
424             //  {
425             //      UINT8           marshalType;        // MIN_MAX_MTYPE
426             //      UINT8           modifiers;
427             //      UINT8           errorCode;
428             //      UINT32          values[2];
429             //  } MinMaxMarshal_mst;
430             UINT32               val;
431         //
432             // A min-max has a range. It can have a bit-field that is indexed to the
433             // min value (something that matches min has a bit at 0. This is useful
434             // for algorithms. The min-max define a range of algorithms to be checked
435             // and the bit field can check to see if the algorithm in that range is
436             // allowed.
437             if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
438                                            buffer, size, &val)))
439             {
440                 MinMaxMarshal_mst   *mmt = (MinMaxMarshal_mst *)sel;
441                 const UINT32        *check = mmt->values;
442             //
443                 // If this type takes a NULL, see if it matches. This
444                 if((mmt->modifiers & TAKES_NULL) && (val == *check++))
445                 {
446                     if((typeIndex & NULL_FLAG) == 0)
447                         result = (TPM_RC)(mmt->errorCode);
448                 }
449                 else
450                 {
451                     val -= *check;
452                     if((val > check[1])
453                        || ((mmt->modifiers & HAS_BITS) &&
454                            !IS_BIT_SET32(val, &check[2])))
455                         result = (TPM_RC)(mmt->errorCode);
456                 }
457             }
458             break;
459         }
460         case ATTRIBUTES_MTYPE:
461         {
462             //  This is used for TPMA values.
```

```
463             UINT32                   mask;
464             AttributesMarshal_mst   *amt = (AttributesMarshal_mst *)sel;
465         //
466             if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
467                                            buffer, size, &mask)))
468             {
469                 if((mask & amt->attributeMask) != 0)
470                     result = TPM_RC_RESERVED_BITS;
471             }
472             break;
473         }
474         case STRUCTURE_MTYPE:
475         {
476             // A structure (not a union). A structure has elements (one defined per
477             // row). Three UINT16 values are used for each row. The first indicates
478             // the type of the entry. They choices are: simple, union, or array. A
479             // simple type can be a simple integer or another structure. It can also
480             // be a specific "interface type." For example, when a structure entry is
481             // a value that is used define the dimension of an array, the entry of
482             // the structure will reference a "synthetic" interface type, most often
483             // a min-max value. If the type of the entry is union or array, then the
484             // first value indicates which of the previous elements provides the union
485             // selector or the array dimension. That previous entry is referenced in
486             // the unmarshaled structure in memory (Not the marshaled buffer). The
487             // previous entry indicates the location in the structure of the value.
488             // The second entry of each structure entry indicated the index of the
489             // type associated with the entry. This is an index into the array of
490             // arrays or the union table (merged with the normal table in this
491             // implementation). The final entry is the offset in the unmarshaled
492             // structure where the value is located. This is the offsetof(STRUcTURE,
493             // element). This value is added to the input 'target' or 'source' value
494             // to determine where the value goes.
495             StructMarshal_mst   *mst = (StructMarshal_mst *)sel;
496             int                  i;
497             const UINT16        *value;
498         //
499             for(result = TPM_RC_SUCCESS, value = mst->values,  i = mst->elements
500                 ; (TPM_RC_SUCCESS == result) && (i > 0)
501                 ; value = &value[3], i--)
502             {
503                 UINT16           descriptor = value[0];
504                 marshalIndex_t   index = value[1];
505                 UINT8           *offset = _target + value[2];
506             //
507                 index |= ((ELEMENT_PROPAGATE & descriptor)
508                             << (NULL_SHIFT - PROPAGATE_SHIFT));
509                 switch(GET_ELEMENT_TYPE(descriptor))
510                 {
511                     case SIMPLE_STYPE:
512                     {
513                         result = Unmarshal(index, offset, buffer, size);
514                         break;
515                     }
516                     case UNION_STYPE:
517                     {
518                         UINT32               choice;
519                     //
520                         // Get the selector or array dimension value
521                         choice = GetSelector(target, mst->values, descriptor);
522                         result = UnmarshalUnion(index, offset, buffer, size, choice);
523                         break;
524                     }
525                     case ARRAY_STYPE:
526                     {
527                         UINT32            dimension;
528                     //
```

```
529                         dimension = GetSelector(target, mst->values, descriptor);
530                         result = ArrayUnmarshal(index, offset, buffer,
531                                                 size, dimension);
532                         break;
533                     }
534                     default:
535                         result = TPM_RC_FAILURE;
536                         break;
537                 }
538             }
539             break;
540         }
541         case TPM2B_MTYPE:
542         {
543             // A primitive TPM2B. A size and byte buffer. The single value (other than
544             // the tag) references the synthetic 'interface' value for the size
545             // parameter.
546             Tpm2bMarshal_mst        *m2bt = (Tpm2bMarshal_mst *)sel;
547         //
548             if(IS_SUCCESS(Unmarshal(m2bt->sizeIndex, target, buffer, size)))
549                 result = UnmarshalBytes(((TPM2B *)target)->buffer,
550                                         buffer, size, *((UINT16 *)target));
551             break;
552         }
553         case TPM2BS_MTYPE:
554         {
555             // This is used when a TPM2B contains a structure.
556             Tpm2bsMarshal_mst   *m2bst = (Tpm2bsMarshal_mst *)sel;
557             INT32          count;
558         //
559             if(IS_SUCCESS(Unmarshal(m2bst->sizeIndex, target, buffer, size)))
560             {
561                 count = (int32_t)*((UINT16 *)_target);
562                 if(count == 0)
563                 {
564                     if(m2bst->modifiers & SIZE_EQUAL)
565                         result = TPM_RC_SIZE;
566                 }
567                 else if((*size -= count) >= 0)
568                 {
569                     marshalIndex_t  index = m2bst->dataIndex;
570                 //
571                     index |= (m2bst->modifiers & PROPAGATE_NULL)
572                                     << (NULL_SHIFT - PROPAGATE_SHIFT);
573                     if(IS_SUCCESS(Unmarshal(index,
574                                             _target + (m2bst->modifiers & SIGNED_MASK),
575                                             buffer, &count)))
576                     {
577                         if(count != 0)
578                             result = TPM_RC_SIZE;
579                     }
580                 }
581                 else
582                     result = TPM_RC_INSUFFICIENT;
583             }
584             break;
585         }
586         case LIST_MTYPE:
587         {
588             // Used for a list. A list is a qualified 32-bit 'count' value followed
589             // by a type indicator.
590             ListMarshal_mst     *mlt = (ListMarshal_mst *)sel;
591             marshalIndex_t      index = mlt->arrayRef;
592         //
593             if(IS_SUCCESS(Unmarshal(mlt->sizeIndex, target, buffer, size)))
594             {
```

```
595                 index |= (mlt->modifiers & PROPAGATE_NULL)
596                             << (NULL_SHIFT - PROPAGATE_SHIFT);
597                 result = ArrayUnmarshal(index,
598                                         _target +(mlt->modifiers & SIGNED_MASK),
599                                         buffer, size,
600                                         *((UINT32 *)target));
601             }
602             break;
603         }
604         case NULL_MTYPE:
605         {
606             result = TPM_RC_SUCCESS;
607             break;
608         }
609         case COMPOSITE_MTYPE:
610         {
611             CompositeMarshal_mst    *mct = (CompositeMarshal_mst *)sel;
612             int                      i;
613             UINT8                   *buf = *buffer;
614             INT32                   sz = *size;
615         //
616             result = TPM_RC_VALUE;
617             for(i = GET_ELEMENT_COUNT(mct->modifiers) - 1; i <= 0; i--)
618             {
619                 marshalIndex_t      index = mct->types[i];
620             //
621                 // This type might take a null so set it in each called value, just
622                 // in case it is needed in that value. Only one value in each
623                 // composite should have the takes null SET.
624                 index |= typeIndex & NULL_MASK;
625                 result = Unmarshal(index, target, buffer, size);
626                 if(result == TPM_RC_SUCCESS)
627                     break;
628                 // Each of the composite values does its own unmarshaling. This
629                 // has some execution overhead if it is unmarshaled multiple times
630                 // but it saves code size in not having to reproduce the various
631                 // unmarshaling types that can be in a composite. So, what this means
632                 // is that the buffer pointer and size have to be reset for each
633                 // unmarshaled value.
634                 *buffer = buf;
635                 *size = sz;
636             }
637             break;
638         }
639         default:
640         {
641             result = TPM_RC_FAILURE;
642             break;
643         }
644     }
645     return result;
646 }
```

9.10.8.1.1.7 Marshal()

This is the function that drives marshaling of output. Because there is no validation of the output, there is a lot less code.

```
647 UINT16 Marshal(
648     UINT16               typeIndex,          // IN: the thing to marshal
649     void                *source,             // IN: were the data comes from
650     UINT8               **buffer,            // IN/OUT: the data source buffer
651     INT32               *size                // IN/OUT: the remaining size
652 )
653 {
654 #define _source  ((UINT8 *)source)
655 
656     const MarshalHeader_mst     *sel;
657     UINT16                       retVal;
658 //
659     sel = GetDescriptor(typeIndex);
660     switch(sel->marshalType)
661     {
662         case VALUES_MTYPE:
663         case UINT_MTYPE:
664         case TABLE_MTYPE:
665         case MIN_MAX_MTYPE:
666         case ATTRIBUTES_MTYPE:
667         case COMPOSITE_MTYPE:
668         {
669 #if BIG_ENDIAN_TPM
670 #define MM16    0
671 #define MM32    0
672 #define MM64    0
673 #else
674 // These flip the constant index values so that they count in reverse order when doing
675 // little-endian stuff
676 #define MM16    1
677 #define MM32    3
678 #define MM64    7
679 #endif
680 // Just change the name and cast the type of the input parameters for typing purposes
681 #define mb (*buffer)
682 #define _source ((UINT8 *)source)
683             retVal = (1 << (sel->modifiers & SIZE_MASK));
684             if(buffer != NULL)
685             {
686                 if((size == NULL) || ((*size -= retVal) >= 0))
687                 {
688                     if(retVal == 4)
689                     {
690                         mb[0 ^ MM32] = _source[0];
691                         mb[1 ^ MM32] = _source[1];
692                         mb[2 ^ MM32] = _source[2];
693                         mb[3 ^ MM32] = _source[3];
694                     }
695                     else if(retVal == 2)
696                     {
697                         mb[0 ^ MM16] = _source[0];
698                         mb[1 ^ MM16] = _source[1];
699                     }
700                     else if(retVal == 1)
701                         mb[0] = _source[0];
702                     else
703                     {
704                         mb[0 ^ MM64] = _source[0];
705                         mb[1 ^ MM64] = _source[1];
```

```
706                         mb[2 ^ MM64] = _source[2];
707                         mb[3 ^ MM64] = _source[3];
708                         mb[4 ^ MM64] = _source[4];
709                         mb[5 ^ MM64] = _source[5];
710                         mb[6 ^ MM64] = _source[6];
711                         mb[7 ^ MM64] = _source[7];
712                     }
713                     *buffer += retVal;
714                 }
715             }
716             break;
717         }
718         case STRUCTURE_MTYPE:
719         {
720 //#define _mst  ((StructMarshal_mst *)sel)
721             StructMarshal_mst *mst = ((StructMarshal_mst *)sel);
722             int             i;
723             const UINT16  *value = mst->values;
724 
725         //
726             for(retVal = 0, i = mst->elements; i > 0; value = &value[3], i--)
727             {
728                 UINT16           des = value[0];
729                 marshalIndex_t   index = value[1];
730                 UINT8           *offset = _source + value[2];
731             //
732                 switch(GET_ELEMENT_TYPE(des))
733                 {
734                     case UNION_STYPE:
735                     {
736                         UINT32           choice;
737                     //
738                         choice = GetSelector(source, mst->values, des);
739                         retVal += MarshalUnion(index, offset, buffer, size, choice);
740                         break;
741                     }
742                     case ARRAY_STYPE:
743                     {
744                         UINT32               count;
745                     //
746                         count = GetSelector(source, mst->values, des);
747                         retVal += ArrayMarshal(index, offset, buffer, size, count);
748                         break;
749                     }
750                     case SIMPLE_STYPE:
751                     default:
752                     {
753                         // This is either another structure or a simple type
754                         retVal += Marshal(index, offset, buffer, size);
755                         break;
756                     }
757                 }
758             }
759             break;
760         }
761         case TPM2B_MTYPE:
762         {
763             // Get the number of bytes being marshaled
764             INT32         val = (int32_t)*((UINT16 *)source);
765         //
766             retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
767 
768             // This is a standard 2B with a byte buffer
769             retVal += MarshalBytes(((TPM2B *)_source)->buffer, buffer, size, val);
770             break;
771         }
```

```
772             case TPM2BS_MTYPE: // A structure in a TPM2B
773             {
774                 Tpm2bsMarshal_mst       *m2bst = (Tpm2bsMarshal_mst *)sel;
775                 UINT8                   *offset;
776                 UINT16                   amount;
777                 UINT8                   *marshaledSize;
778             //
779                 // Save the address of where the size should go
780                 marshaledSize = *buffer;
781
782                 // marshal the size (checks the space and advanced the pointer)
783                 retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
784
785                 // This gets the offsetof the structure to marshal. It was placed in the
786                 // modifiers byte because the offset from the start of the TPM2B to the
787                 // start of the structure is going to be less than 8 and the modifiers
788                 // byte isn't needed for anything else.
789                 offset = _source + (m2bst->modifiers & SIGNED_MASK);
790
791                 // Marshal the structure and get its size
792                 amount = Marshal(m2bst->dataIndex, offset, buffer, size);
793
794                 // put the size in the space used when the size was marshaled.
795                 if(buffer != NULL)
796                     UINT16_TO_BYTE_ARRAY(amount, marshaledSize);
797                 retVal += amount;
798                 break;
799             }
800             case LIST_MTYPE:
801             {
802                 ListMarshal_mst *    mlt = ((ListMarshal_mst *)sel);
803                 UINT8               *offset = _source + (mlt->modifiers & SIGNED_MASK);
804                 retVal = Marshal(UINT32_MARSHAL_REF, source, buffer, size);
805                 retVal += ArrayMarshal((marshalIndex_t)(mlt->arrayRef), offset,
806                                        buffer, size, *((UINT32 *)source));
807                 break;
808             }
809             case NULL_MTYPE:
810                 retVal = 0;
811                 break;
812           case ERROR_MTYPE:
813           default:
814             {
815                 if(size != NULL)
816                    *size = -1;
817                 retVal = 0;
818                 break;
819             }
820         }
821         return retVal;
822
823     }
824     #endif // TABLE_DRIVEN_MARSHAL
```

9.10.8.2 TableMarshalData.c

This file contains the data initializer used for the table-driven marshaling code.

```
1   #include "Tpm.h"
2   #if TABLE_DRIVEN_MARSHAL
3   #include "TableMarshal.h"
4   #include "Marshal.h"
```

The array marshaling table

```
ArrayMarshal_mst   ArrayLookupTable[] = {
    ARRAY_MARSHAL_ENTRY(UINT8),
    ARRAY_MARSHAL_ENTRY(TPM_CC),
    ARRAY_MARSHAL_ENTRY(TPMA_CC),
    ARRAY_MARSHAL_ENTRY(TPM_ALG_ID),
    ARRAY_MARSHAL_ENTRY(TPM_HANDLE),
    ARRAY_MARSHAL_ENTRY(TPM2B_DIGEST),
    ARRAY_MARSHAL_ENTRY(TPMT_HA),
    ARRAY_MARSHAL_ENTRY(TPMS_PCR_SELECTION),
    ARRAY_MARSHAL_ENTRY(TPMS_ALG_PROPERTY),
    ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PROPERTY),
    ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PCR_SELECT),
    ARRAY_MARSHAL_ENTRY(TPM_ECC_CURVE),
    ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_POLICY),
    ARRAY_MARSHAL_ENTRY(TPMS_ACT_DATA),
    ARRAY_MARSHAL_ENTRY(TPMS_AC_OUTPUT)};
```

The main marshaling structure

```
MarshalData_st MarshalData = {
// UINT8_DATA
{UINT_MTYPE, 0},
// UINT16_DATA
{UINT_MTYPE, 1},
// UINT32_DATA
{UINT_MTYPE, 2},
// UINT64_DATA
{UINT_MTYPE, 3},
// INT8_DATA
{UINT_MTYPE, 0 + IS_SIGNED},
// INT16_DATA
{UINT_MTYPE, 1 + IS_SIGNED},
// INT32_DATA
{UINT_MTYPE, 2 + IS_SIGNED},
// INT64_DATA
{UINT_MTYPE, 3 + IS_SIGNED},
// UINT0_DATA
{NULL_MTYPE, 0},
// TPM_ECC_CURVE_DATA
{MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_CURVE,
    {TPM_ECC_NONE,
    RANGE(1, 32, UINT16),
    ((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
     (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15)  |
     (ECC_BN_P638 << 16)  | (ECC_SM2_P256 << 31))}},
// TPM_CLOCK_ADJUST_DATA
{MIN_MAX_MTYPE, ONE_BYTES|IS_SIGNED, (UINT8)TPM_RC_VALUE,
    {RANGE(TPM_CLOCK_COARSE_SLOWER, TPM_CLOCK_COARSE_FASTER, INT8)}},
// TPM_EO_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE,
    {RANGE(TPM_EO_EQ, TPM_EO_BITCLEAR, UINT16)}},
// TPM_SU_DATA
{TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 2,
    {TPM_SU_CLEAR, TPM_SU_STATE}},
// TPM_SE_DATA
{TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 3,
    {TPM_SE_HMAC, TPM_SE_POLICY, TPM_SE_TRIAL}},
// TPM_CAP_DATA
{VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
   {RANGE(TPM_CAP_ALGS, TPM_CAP_ACT, UINT32),
    TPM_CAP_VENDOR_PROPERTY}},
// TPMA_ALGORITHM_DATA
{ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFF8F0},
// TPMA_OBJECT_DATA
{ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFF0F309},
```

```
67  // TPMA_SESSION_DATA
68  {ATTRIBUTES_MTYPE, ONE_BYTES, 0x00000018},
69  // TPMA_ACT_DATA
70  {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFFFFC},
71  // TPMI_YES_NO_DATA
72  {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 2,
73      {NO, YES}},
74  // TPMI_DH_OBJECT_DATA
75  {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
76      {TPM_RH_NULL,
77      RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
78      RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
79  // TPMI_DH_PARENT_DATA
80  {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 3,
81      {TPM_RH_NULL,
82      RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
83      RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
84      TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
85  // TPMI_DH_PERSISTENT_DATA
86  {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
87      {RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
88  // TPMI_DH_ENTITY_DATA
89  {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 5, 4,
90      {TPM_RH_NULL,
91      RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
92      RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
93      RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
94      RANGE(PCR_FIRST, PCR_LAST, UINT32),
95      RANGE(TPM_RH_AUTH_00, TPM_RH_AUTH_FF, UINT32),
96      TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
97  // TPMI_DH_PCR_DATA
98  {MIN_MAX_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE,
99      {TPM_RH_NULL,
100     RANGE(PCR_FIRST, PCR_LAST, UINT32)}},
101 // TPMI_SH_AUTH_SESSION_DATA
102 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
103     {TPM_RS_PW,
104     RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
105     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
106 // TPMI_SH_HMAC_DATA
107 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
108     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32)}},
109 // TPMI_SH_POLICY_DATA
110 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
111     {RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
112 // TPMI_DH_CONTEXT_DATA
113 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 3, 0,
114     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
115     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
116     RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32)}},
117 // TPMI_DH_SAVED_DATA
118 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2, 3,
119     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
120     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
121     0x80000000, 0x80000001, 0x80000002}},
122 // TPMI_RH_HIERARCHY_DATA
123 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 3,
124     {TPM_RH_NULL,
125     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
126 // TPMI_RH_ENABLES_DATA
127 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 4,
128     {TPM_RH_NULL,
129     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM, TPM_RH_PLATFORM_NV}},
130 // TPMI_RH_HIERARCHY_AUTH_DATA
131 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4,
132     {TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
```

```
133   // TPMI_RH_HIERARCHY_POLICY_DATA
134   {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 4,
135       {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32),
136       TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
137   // TPMI_RH_PLATFORM_DATA
138   {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
139       {TPM_RH_PLATFORM}},
140   // TPMI_RH_OWNER_DATA
141   {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
142       {TPM_RH_NULL,
143       TPM_RH_OWNER}},
144   // TPMI_RH_ENDORSEMENT_DATA
145   {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
146       {TPM_RH_NULL,
147       TPM_RH_ENDORSEMENT}},
148   // TPMI_RH_PROVISION_DATA
149   {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
150       {TPM_RH_OWNER, TPM_RH_PLATFORM}},
151   // TPMI_RH_CLEAR_DATA
152   {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
153       {TPM_RH_LOCKOUT, TPM_RH_PLATFORM}},
154   // TPMI_RH_NV_AUTH_DATA
155   {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 2,
156       {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
157       TPM_RH_OWNER, TPM_RH_PLATFORM}},
158   // TPMI_RH_LOCKOUT_DATA
159   {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
160       {TPM_RH_LOCKOUT}},
161   // TPMI_RH_NV_INDEX_DATA
162   {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
163       {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32)}},
164   // TPMI_RH_AC_DATA
165   {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
166       {RANGE(AC_FIRST, AC_LAST, UINT32)}},
167   // TPMI_RH_ACT_DATA
168   {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
169       {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32)}},
170   // TPMI_ALG_HASH_DATA
171   {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_HASH,
172       {TPM_ALG_NULL,
173       RANGE(4, 41, UINT16),
174       ((ALG_SHA1 << 0)    | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
175        (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
176       ((ALG_SHA3_256 << 3)|(ALG_SHA3_384 << 4)|(ALG_SHA3_512 << 5))}},
177   // TPMI_ALG_ASYM_DATA
178   {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_ASYMMETRIC,
179       {TPM_ALG_NULL,
180       RANGE(1, 35, UINT16),
181       ((ALG_RSA << 0)),
182       ((ALG_ECC << 2))}},
183   // TPMI_ALG_SYM_DATA
184   {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
185       {TPM_ALG_NULL,
186       RANGE(3, 38, UINT16),
187       ((ALG_TDES << 0)|(ALG_AES << 3)|(ALG_XOR << 7)|(ALG_SM4 << 16)),
188       ((ALG_CAMELLIA << 3))}},
189   // TPMI_ALG_SYM_OBJECT_DATA
190   {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
191       {TPM_ALG_NULL,
192       RANGE(3, 38, UINT16),
193       ((ALG_TDES << 0)|(ALG_AES << 3)|(ALG_SM4 << 16)),
194       ((ALG_CAMELLIA << 3))}},
195   // TPMI_ALG_SYM_MODE_DATA
196   {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
197       {TPM_ALG_NULL,
198       RANGE(63, 68, UINT16),
```

```
199         ((ALG_CMAC << 0) | (ALG_CTR << 1)  | (ALG_OFB << 2)  | (ALG_CBC << 3)  |
200          (ALG_CFB << 4)  | (ALG_ECB << 5))}},
201     // TPMI_ALG_KDF_DATA
202     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_KDF,
203         {TPM_ALG_NULL,
204         RANGE(7, 34, UINT16),
205         ((ALG_MGF1 << 0)            | (ALG_KDF1_SP800_56A << 25) |
206          (ALG_KDF2 << 26)           | (ALG_KDF1_SP800_108 << 27))}},
207     // TPMI_ALG_SIG_SCHEME_DATA
208     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
209         {TPM_ALG_NULL,
210         RANGE(5, 28, UINT16),
211         ((ALG_HMAC << 0)    | (ALG_RSASSA << 15) | (ALG_RSAPSS << 17) |
212          (ALG_ECDSA << 19)  | (ALG_ECDAA << 21)  | (ALG_SM2 << 22)    |
213          (ALG_ECSCHNORR << 23))}},
214     // TPMI_ECC_KEY_EXCHANGE_DATA
215     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
216         {TPM_ALG_NULL,
217         RANGE(25, 29, UINT16),
218         ((ALG_ECDH << 0)|(ALG_SM2 << 2)|(ALG_ECMQV << 4))}},
219     // TPMI_ST_COMMAND_TAG_DATA
220     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_BAD_TAG, 2,
221         {TPM_ST_NO_SESSIONS, TPM_ST_SESSIONS}},
222     // TPMI_ALG_MAC_SCHEME_DATA
223     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
224         {TPM_ALG_NULL,
225         RANGE(4, 63, UINT16),
226         ((ALG_SHA1 << 0)    | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
227          (ALG_SHA512 << 9)  | (ALG_SM3_256 << 14)),
228         ((ALG_SHA3_256 << 3)|(ALG_SHA3_384 << 4)|(ALG_SHA3_512 << 5)|(ALG_CMAC << 27))}},
229     // TPMI_ALG_CIPHER_MODE_DATA
230     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
231         {TPM_ALG_NULL,
232         RANGE(64, 68, UINT16),
233         ((ALG_CTR << 0)|(ALG_OFB << 1)|(ALG_CBC << 2)|(ALG_CFB << 3)|(ALG_ECB << 4))}},
234     // TPMS_EMPTY_DATA
235     {STRUCTURE_MTYPE, 1,
236          {SET_ELEMENT_TYPE(SIMPLE_STYPE), UINT0_MARSHAL_REF, 0}},
237     // TPMS_ALGORITHM_DESCRIPTION_DATA
238     {STRUCTURE_MTYPE, 2, {
239             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
240                 TPM_ALG_ID_MARSHAL_REF,
241                 (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, alg)),
242             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
243                 TPMA_ALGORITHM_MARSHAL_REF,
244                 (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, attributes))}},
245     // TPMU_HA_DATA
246     {9, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_HA_mst, marshalingTypes)),
247         {(UINT32)TPM_ALG_SHA1,     (UINT32)TPM_ALG_SHA256,   (UINT32)TPM_ALG_SHA384,
248          (UINT32)TPM_ALG_SHA512,   (UINT32)TPM_ALG_SM3_256,  (UINT32)TPM_ALG_SHA3_256,
249          (UINT32)TPM_ALG_SHA3_384, (UINT32)TPM_ALG_SHA3_512, (UINT32)TPM_ALG_NULL},
250         {(UINT16)(SHA1_DIGEST_SIZE),     (UINT16)(SHA256_DIGEST_SIZE),
251          (UINT16)(SHA384_DIGEST_SIZE),   (UINT16)(SHA512_DIGEST_SIZE),
252          (UINT16)(SM3_256_DIGEST_SIZE),  (UINT16)(SHA3_256_DIGEST_SIZE),
253          (UINT16)(SHA3_384_DIGEST_SIZE), (UINT16)(SHA3_512_DIGEST_SIZE),
254          (UINT16)(0)}
255     },
256     // TPMT_HA_DATA
257     {STRUCTURE_MTYPE, 2, {
258             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
259                 TPMI_ALG_HASH_MARSHAL_REF,
260                 (UINT16)(offsetof(TPMT_HA, hashAlg)),
261             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
262                 TPMU_HA_MARSHAL_REF,
263                 (UINT16)(offsetof(TPMT_HA, digest))}},
264     // TPM2B_DIGEST_DATA
```

```
265     {TPM2B_MTYPE, Type00_MARSHAL_REF},
266     // TPM2B_DATA_DATA
267     {TPM2B_MTYPE, Type01_MARSHAL_REF},
268     // TPM2B_EVENT_DATA
269     {TPM2B_MTYPE, Type02_MARSHAL_REF},
270     // TPM2B_MAX_BUFFER_DATA
271     {TPM2B_MTYPE, Type03_MARSHAL_REF},
272     // TPM2B_MAX_NV_BUFFER_DATA
273     {TPM2B_MTYPE, Type04_MARSHAL_REF},
274     // TPM2B_TIMEOUT_DATA
275     {TPM2B_MTYPE, Type05_MARSHAL_REF},
276     // TPM2B_IV_DATA
277     {TPM2B_MTYPE, Type06_MARSHAL_REF},
278     // NULL_UNION_DATA
279     {0},
280     // TPM2B_NAME_DATA
281     {TPM2B_MTYPE, Type07_MARSHAL_REF},
282     // TPMS_PCR_SELECT_DATA
283     {STRUCTURE_MTYPE, 2, {
284             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
285                 Type08_MARSHAL_REF,
286                 (UINT16)(offsetof(TPMS_PCR_SELECT, sizeofSelect)),
287             SET_ELEMENT_TYPE(ARRAY_STYPE)|SET_ELEMENT_NUMBER(0),
288                 UINT8_ARRAY_MARSHAL_INDEX,
289                 (UINT16)(offsetof(TPMS_PCR_SELECT, pcrSelect))}},
290     // TPMS_PCR_SELECTION_DATA
291     {STRUCTURE_MTYPE, 3, {
292             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
293                 TPMI_ALG_HASH_MARSHAL_REF,
294                 (UINT16)(offsetof(TPMS_PCR_SELECTION, hash)),
295             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
296                 Type08_MARSHAL_REF,
297                 (UINT16)(offsetof(TPMS_PCR_SELECTION, sizeofSelect)),
298             SET_ELEMENT_TYPE(ARRAY_STYPE)|SET_ELEMENT_NUMBER(1),
299                 UINT8_ARRAY_MARSHAL_INDEX,
300                 (UINT16)(offsetof(TPMS_PCR_SELECTION, pcrSelect))}},
301     // TPMT_TK_CREATION_DATA
302     {STRUCTURE_MTYPE, 3, {
303             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
304                 Type10_MARSHAL_REF,
305                 (UINT16)(offsetof(TPMT_TK_CREATION, tag)),
306             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
307                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
308                 (UINT16)(offsetof(TPMT_TK_CREATION, hierarchy)),
309             SET_ELEMENT_TYPE(SIMPLE_STYPE),
310                 TPM2B_DIGEST_MARSHAL_REF,
311                 (UINT16)(offsetof(TPMT_TK_CREATION, digest))}},
312     // TPMT_TK_VERIFIED_DATA
313     {STRUCTURE_MTYPE, 3, {
314             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
315                 Type11_MARSHAL_REF,
316                 (UINT16)(offsetof(TPMT_TK_VERIFIED, tag)),
317             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
318                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
319                 (UINT16)(offsetof(TPMT_TK_VERIFIED, hierarchy)),
320             SET_ELEMENT_TYPE(SIMPLE_STYPE),
321                 TPM2B_DIGEST_MARSHAL_REF,
322                 (UINT16)(offsetof(TPMT_TK_VERIFIED, digest))}},
323     // TPMT_TK_AUTH_DATA
324     {STRUCTURE_MTYPE, 3, {
325             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
326                 Type12_MARSHAL_REF,
327                 (UINT16)(offsetof(TPMT_TK_AUTH, tag)),
328             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
329                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
330                 (UINT16)(offsetof(TPMT_TK_AUTH, hierarchy)),
```

```
331             SET_ELEMENT_TYPE(SIMPLE_STYPE),
332                 TPM2B_DIGEST_MARSHAL_REF,
333                 (UINT16)(offsetof(TPMT_TK_AUTH, digest))},
334     // TPMT_TK_HASHCHECK_DATA
335     {STRUCTURE_MTYPE, 3, {
336             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
337                 Type13_MARSHAL_REF,
338                 (UINT16)(offsetof(TPMT_TK_HASHCHECK, tag)),
339             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
340                 TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
341                 (UINT16)(offsetof(TPMT_TK_HASHCHECK, hierarchy)),
342             SET_ELEMENT_TYPE(SIMPLE_STYPE),
343                 TPM2B_DIGEST_MARSHAL_REF,
344                 (UINT16)(offsetof(TPMT_TK_HASHCHECK, digest))},
345     // TPMS_ALG_PROPERTY_DATA
346     {STRUCTURE_MTYPE, 2, {
347             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
348                 TPM_ALG_ID_MARSHAL_REF,
349                 (UINT16)(offsetof(TPMS_ALG_PROPERTY, alg)),
350             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
351                 TPMA_ALGORITHM_MARSHAL_REF,
352                 (UINT16)(offsetof(TPMS_ALG_PROPERTY, algProperties))},
353     // TPMS_TAGGED_PROPERTY_DATA
354     {STRUCTURE_MTYPE, 2, {
355             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
356                 TPM_PT_MARSHAL_REF,
357                 (UINT16)(offsetof(TPMS_TAGGED_PROPERTY, property)),
358             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
359                 UINT32_MARSHAL_REF,
360                 (UINT16)(offsetof(TPMS_TAGGED_PROPERTY, value))},
361     // TPMS_TAGGED_PCR_SELECT_DATA
362     {STRUCTURE_MTYPE, 3, {
363             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
364                 TPM_PT_PCR_MARSHAL_REF,
365                 (UINT16)(offsetof(TPMS_TAGGED_PCR_SELECT, tag)),
366             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
367                 Type08_MARSHAL_REF,
368                 (UINT16)(offsetof(TPMS_TAGGED_PCR_SELECT, sizeofSelect)),
369             SET_ELEMENT_TYPE(ARRAY_STYPE)|SET_ELEMENT_NUMBER(1),
370                 UINT8_ARRAY_MARSHAL_INDEX,
371                 (UINT16)(offsetof(TPMS_TAGGED_PCR_SELECT, pcrSelect))},
372     // TPMS_TAGGED_POLICY_DATA
373     {STRUCTURE_MTYPE, 2, {
374             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
375                 TPM_HANDLE_MARSHAL_REF,
376                 (UINT16)(offsetof(TPMS_TAGGED_POLICY, handle)),
377             SET_ELEMENT_TYPE(SIMPLE_STYPE),
378                 TPMT_HA_MARSHAL_REF,
379                 (UINT16)(offsetof(TPMS_TAGGED_POLICY, policyHash))},
380     // TPMS_ACT_DATA_DATA
381     {STRUCTURE_MTYPE, 3, {
382             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
383                 TPM_HANDLE_MARSHAL_REF,
384                 (UINT16)(offsetof(TPMS_ACT_DATA, handle)),
385             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
386                 UINT32_MARSHAL_REF,
387                 (UINT16)(offsetof(TPMS_ACT_DATA, timeout)),
388             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
389                 TPMA_ACT_MARSHAL_REF,
390                 (UINT16)(offsetof(TPMS_ACT_DATA, attributes))},
391     // TPML_CC_DATA
392     {LIST_MTYPE,
393         (UINT8)(offsetof(TPML_CC, commandCodes)),
394         Type15_MARSHAL_REF,
395         TPM_CC_ARRAY_MARSHAL_INDEX},
396     // TPML_CCA_DATA
```

```
397   {LIST_MTYPE,
398       (UINT8)(offsetof(TPML_CCA, commandAttributes)),
399       Type15_MARSHAL_REF,
400       TPMA_CC_ARRAY_MARSHAL_INDEX},
401   // TPML_ALG_DATA
402   {LIST_MTYPE,
403       (UINT8)(offsetof(TPML_ALG, algorithms)),
404       Type17_MARSHAL_REF,
405       TPM_ALG_ID_ARRAY_MARSHAL_INDEX},
406   // TPML_HANDLE_DATA
407   {LIST_MTYPE,
408       (UINT8)(offsetof(TPML_HANDLE, handle)),
409       Type18_MARSHAL_REF,
410       TPM_HANDLE_ARRAY_MARSHAL_INDEX},
411   // TPML_DIGEST_DATA
412   {LIST_MTYPE,
413       (UINT8)(offsetof(TPML_DIGEST, digests)),
414       Type19_MARSHAL_REF,
415       TPM2B_DIGEST_ARRAY_MARSHAL_INDEX},
416   // TPML_DIGEST_VALUES_DATA
417   {LIST_MTYPE,
418       (UINT8)(offsetof(TPML_DIGEST_VALUES, digests)),
419       Type20_MARSHAL_REF,
420       TPMT_HA_ARRAY_MARSHAL_INDEX},
421   // TPML_PCR_SELECTION_DATA
422   {LIST_MTYPE,
423       (UINT8)(offsetof(TPML_PCR_SELECTION, pcrSelections)),
424       Type20_MARSHAL_REF,
425       TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX},
426   // TPML_ALG_PROPERTY_DATA
427   {LIST_MTYPE,
428       (UINT8)(offsetof(TPML_ALG_PROPERTY, algProperties)),
429       Type22_MARSHAL_REF,
430       TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX},
431   // TPML_TAGGED_TPM_PROPERTY_DATA
432   {LIST_MTYPE,
433       (UINT8)(offsetof(TPML_TAGGED_TPM_PROPERTY, tpmProperty)),
434       Type23_MARSHAL_REF,
435       TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX},
436   // TPML_TAGGED_PCR_PROPERTY_DATA
437   {LIST_MTYPE,
438       (UINT8)(offsetof(TPML_TAGGED_PCR_PROPERTY, pcrProperty)),
439       Type24_MARSHAL_REF,
440       TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX},
441   // TPML_ECC_CURVE_DATA
442   {LIST_MTYPE,
443       (UINT8)(offsetof(TPML_ECC_CURVE, eccCurves)),
444       Type25_MARSHAL_REF,
445       TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX},
446   // TPML_TAGGED_POLICY_DATA
447   {LIST_MTYPE,
448       (UINT8)(offsetof(TPML_TAGGED_POLICY, policies)),
449       Type26_MARSHAL_REF,
450       TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX},
451   // TPML_ACT_DATA_DATA
452   {LIST_MTYPE,
453       (UINT8)(offsetof(TPML_ACT_DATA, actData)),
454       Type27_MARSHAL_REF,
455       TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX},
456   // TPMU_CAPABILITIES_DATA
457   {11, 0, (UINT16)(offsetof(TPMU_CAPABILITIES_mst, marshalingTypes)),
458       {(UINT32)TPM_CAP_ALGS,           (UINT32)TPM_CAP_HANDLES,
459        (UINT32)TPM_CAP_COMMANDS,       (UINT32)TPM_CAP_PP_COMMANDS,
460        (UINT32)TPM_CAP_AUDIT_COMMANDS, (UINT32)TPM_CAP_PCRS,
461        (UINT32)TPM_CAP_TPM_PROPERTIES, (UINT32)TPM_CAP_PCR_PROPERTIES,
462        (UINT32)TPM_CAP_ECC_CURVES,     (UINT32)TPM_CAP_AUTH_POLICIES,
```

```
463         (UINT32)TPM_CAP_ACT},
464       {(UINT16)(TPML_ALG_PROPERTY_MARSHAL_REF),
465         (UINT16)(TPML_HANDLE_MARSHAL_REF),
466         (UINT16)(TPML_CCA_MARSHAL_REF),
467         (UINT16)(TPML_CC_MARSHAL_REF),
468         (UINT16)(TPML_CC_MARSHAL_REF),
469         (UINT16)(TPML_PCR_SELECTION_MARSHAL_REF),
470         (UINT16)(TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF),
471         (UINT16)(TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF),
472         (UINT16)(TPML_ECC_CURVE_MARSHAL_REF),
473         (UINT16)(TPML_TAGGED_POLICY_MARSHAL_REF),
474         (UINT16)(TPML_ACT_DATA_MARSHAL_REF)}
475   },
476   // TPMS_CAPABILITY_DATA_DATA
477   {STRUCTURE_MTYPE, 2, {
478           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
479               TPM_CAP_MARSHAL_REF,
480               (UINT16)(offsetof(TPMS_CAPABILITY_DATA, capability)),
481           SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
482               TPMU_CAPABILITIES_MARSHAL_REF,
483               (UINT16)(offsetof(TPMS_CAPABILITY_DATA, data))}},
484   // TPMS_CLOCK_INFO_DATA
485   {STRUCTURE_MTYPE, 4, {
486           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
487               UINT64_MARSHAL_REF,
488               (UINT16)(offsetof(TPMS_CLOCK_INFO, clock)),
489           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
490               UINT32_MARSHAL_REF,
491               (UINT16)(offsetof(TPMS_CLOCK_INFO, resetCount)),
492           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
493               UINT32_MARSHAL_REF,
494               (UINT16)(offsetof(TPMS_CLOCK_INFO, restartCount)),
495           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
496               TPMI_YES_NO_MARSHAL_REF,
497               (UINT16)(offsetof(TPMS_CLOCK_INFO, safe))}},
498   // TPMS_TIME_INFO_DATA
499   {STRUCTURE_MTYPE, 2, {
500           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
501               UINT64_MARSHAL_REF,
502               (UINT16)(offsetof(TPMS_TIME_INFO, time)),
503           SET_ELEMENT_TYPE(SIMPLE_STYPE),
504               TPMS_CLOCK_INFO_MARSHAL_REF,
505               (UINT16)(offsetof(TPMS_TIME_INFO, clockInfo))}},
506   // TPMS_TIME_ATTEST_INFO_DATA
507   {STRUCTURE_MTYPE, 2, {
508           SET_ELEMENT_TYPE(SIMPLE_STYPE),
509               TPMS_TIME_INFO_MARSHAL_REF,
510               (UINT16)(offsetof(TPMS_TIME_ATTEST_INFO, time)),
511           SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
512               UINT64_MARSHAL_REF,
513               (UINT16)(offsetof(TPMS_TIME_ATTEST_INFO, firmwareVersion))}},
514   // TPMS_CERTIFY_INFO_DATA
515   {STRUCTURE_MTYPE, 2, {
516           SET_ELEMENT_TYPE(SIMPLE_STYPE),
517               TPM2B_NAME_MARSHAL_REF,
518               (UINT16)(offsetof(TPMS_CERTIFY_INFO, name)),
519           SET_ELEMENT_TYPE(SIMPLE_STYPE),
520               TPM2B_NAME_MARSHAL_REF,
521               (UINT16)(offsetof(TPMS_CERTIFY_INFO, qualifiedName))}},
522   // TPMS_QUOTE_INFO_DATA
523   {STRUCTURE_MTYPE, 2, {
524           SET_ELEMENT_TYPE(SIMPLE_STYPE),
525               TPML_PCR_SELECTION_MARSHAL_REF,
526               (UINT16)(offsetof(TPMS_QUOTE_INFO, pcrSelect)),
527           SET_ELEMENT_TYPE(SIMPLE_STYPE),
528               TPM2B_DIGEST_MARSHAL_REF,
```

```
            (UINT16)(offsetof(TPMS_QUOTE_INFO, pcrDigest))},
// TPMS_COMMAND_AUDIT_INFO_DATA
{STRUCTURE_MTYPE, 4, {
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
            UINT64_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, auditCounter)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
            TPM_ALG_ID_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, digestAlg)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DIGEST_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, auditDigest)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DIGEST_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_COMMAND_AUDIT_INFO, commandDigest))}},
// TPMS_SESSION_AUDIT_INFO_DATA
{STRUCTURE_MTYPE, 2, {
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
            TPMI_YES_NO_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_SESSION_AUDIT_INFO, exclusiveSession)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DIGEST_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_SESSION_AUDIT_INFO, sessionDigest))}},
// TPMS_CREATION_INFO_DATA
{STRUCTURE_MTYPE, 2, {
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_NAME_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_INFO, objectName)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DIGEST_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_INFO, creationHash))}},
// TPMS_NV_CERTIFY_INFO_DATA
{STRUCTURE_MTYPE, 3, {
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_NAME_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_NV_CERTIFY_INFO, indexName)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
            UINT16_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_NV_CERTIFY_INFO, offset)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_MAX_NV_BUFFER_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_NV_CERTIFY_INFO, nvContents))}},
// TPMS_NV_DIGEST_CERTIFY_INFO_DATA
{STRUCTURE_MTYPE, 2, {
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_NAME_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, indexName)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DIGEST_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, nvDigest))}},
// TPMI_ST_ATTEST_DATA
{VALUES_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
   {RANGE(TPM_ST_ATTEST_NV, TPM_ST_ATTEST_CREATION, UINT16),
    TPM_ST_ATTEST_NV_DIGEST}},
// TPMU_ATTEST_DATA
{8, 0, (UINT16)(offsetof(TPMU_ATTEST_mst, marshalingTypes)),
    {(UINT32)TPM_ST_ATTEST_CERTIFY,       (UINT32)TPM_ST_ATTEST_CREATION,
     (UINT32)TPM_ST_ATTEST_QUOTE,         (UINT32)TPM_ST_ATTEST_COMMAND_AUDIT,
     (UINT32)TPM_ST_ATTEST_SESSION_AUDIT, (UINT32)TPM_ST_ATTEST_TIME,
     (UINT32)TPM_ST_ATTEST_NV,            (UINT32)TPM_ST_ATTEST_NV_DIGEST},
    {(UINT16)(TPMS_CERTIFY_INFO_MARSHAL_REF),
     (UINT16)(TPMS_CREATION_INFO_MARSHAL_REF),
     (UINT16)(TPMS_QUOTE_INFO_MARSHAL_REF),
     (UINT16)(TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF),
     (UINT16)(TPMS_SESSION_AUDIT_INFO_MARSHAL_REF),
     (UINT16)(TPMS_TIME_ATTEST_INFO_MARSHAL_REF),
```

```
595         (UINT16)(TPMS_NV_CERTIFY_INFO_MARSHAL_REF),
596         (UINT16)(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF)}
597     },
598     // TPMS_ATTEST_DATA
599     {STRUCTURE_MTYPE, 7, {
600             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
601                 TPM_GENERATED_MARSHAL_REF,
602                 (UINT16)(offsetof(TPMS_ATTEST, magic)),
603             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
604                 TPMI_ST_ATTEST_MARSHAL_REF,
605                 (UINT16)(offsetof(TPMS_ATTEST, type)),
606             SET_ELEMENT_TYPE(SIMPLE_STYPE),
607                 TPM2B_NAME_MARSHAL_REF,
608                 (UINT16)(offsetof(TPMS_ATTEST, qualifiedSigner)),
609             SET_ELEMENT_TYPE(SIMPLE_STYPE),
610                 TPM2B_DATA_MARSHAL_REF,
611                 (UINT16)(offsetof(TPMS_ATTEST, extraData)),
612             SET_ELEMENT_TYPE(SIMPLE_STYPE),
613                 TPMS_CLOCK_INFO_MARSHAL_REF,
614                 (UINT16)(offsetof(TPMS_ATTEST, clockInfo)),
615             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
616                 UINT64_MARSHAL_REF,
617                 (UINT16)(offsetof(TPMS_ATTEST, firmwareVersion)),
618             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(1),
619                 TPMU_ATTEST_MARSHAL_REF,
620                 (UINT16)(offsetof(TPMS_ATTEST, attested))}},
621     // TPM2B_ATTEST_DATA
622     {TPM2B_MTYPE, Type28_MARSHAL_REF},
623     // TPMS_AUTH_COMMAND_DATA
624     {STRUCTURE_MTYPE, 4, {
625             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
626                 TPMI_SH_AUTH_SESSION_MARSHAL_REF|NULL_FLAG,
627                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, sessionHandle)),
628             SET_ELEMENT_TYPE(SIMPLE_STYPE),
629                 TPM2B_NONCE_MARSHAL_REF,
630                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, nonce)),
631             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
632                 TPMA_SESSION_MARSHAL_REF,
633                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, sessionAttributes)),
634             SET_ELEMENT_TYPE(SIMPLE_STYPE),
635                 TPM2B_AUTH_MARSHAL_REF,
636                 (UINT16)(offsetof(TPMS_AUTH_COMMAND, hmac))}},
637     // TPMS_AUTH_RESPONSE_DATA
638     {STRUCTURE_MTYPE, 3, {
639             SET_ELEMENT_TYPE(SIMPLE_STYPE),
640                 TPM2B_NONCE_MARSHAL_REF,
641                 (UINT16)(offsetof(TPMS_AUTH_RESPONSE, nonce)),
642             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
643                 TPMA_SESSION_MARSHAL_REF,
644                 (UINT16)(offsetof(TPMS_AUTH_RESPONSE, sessionAttributes)),
645             SET_ELEMENT_TYPE(SIMPLE_STYPE),
646                 TPM2B_AUTH_MARSHAL_REF,
647                 (UINT16)(offsetof(TPMS_AUTH_RESPONSE, hmac))}},
648     // TPMI_TDES_KEY_BITS_DATA
649     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
650         {128*TDES_128}},
651     // TPMI_AES_KEY_BITS_DATA
652     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
653         {192*AES_192, 128*AES_128, 256*AES_256}},
654     // TPMI_SM4_KEY_BITS_DATA
655     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
656         {128*SM4_128}},
657     // TPMI_CAMELLIA_KEY_BITS_DATA
658     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
659         {192*CAMELLIA_192, 128*CAMELLIA_128, 256*CAMELLIA_256}},
660     // TPMU_SYM_KEY_BITS_DATA
```

```
661     {6, 0, (UINT16)(offsetof(TPMU_SYM_KEY_BITS_mst, marshalingTypes)),
662         {(UINT32)TPM_ALG_TDES,     (UINT32)TPM_ALG_AES,      (UINT32)TPM_ALG_SM4,
663          (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR,      (UINT32)TPM_ALG_NULL},
664         {(UINT16)(TPMI_TDES_KEY_BITS_MARSHAL_REF),
665          (UINT16)(TPMI_AES_KEY_BITS_MARSHAL_REF),
666          (UINT16)(TPMI_SM4_KEY_BITS_MARSHAL_REF),
667          (UINT16)(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF),
668          (UINT16)(TPMI_ALG_HASH_MARSHAL_REF),
669          (UINT16)(UINT0_MARSHAL_REF)}
670     },
671     // TPMU_SYM_MODE_DATA
672     {6, 0, (UINT16)(offsetof(TPMU_SYM_MODE_mst, marshalingTypes)),
673         {(UINT32)TPM_ALG_TDES,     (UINT32)TPM_ALG_AES,      (UINT32)TPM_ALG_SM4,
674          (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR,      (UINT32)TPM_ALG_NULL},
675         {(UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
676          (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
677          (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
678          (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
679          (UINT16)(UINT0_MARSHAL_REF),
680          (UINT16)(UINT0_MARSHAL_REF)}
681     },
682     // TPMT_SYM_DEF_DATA
683     {STRUCTURE_MTYPE, 3, {
684             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
685                 TPMI_ALG_SYM_MARSHAL_REF,
686                 (UINT16)(offsetof(TPMT_SYM_DEF, algorithm)),
687             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
688                 TPMU_SYM_KEY_BITS_MARSHAL_REF,
689                 (UINT16)(offsetof(TPMT_SYM_DEF, keyBits)),
690             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
691                 TPMU_SYM_MODE_MARSHAL_REF,
692                 (UINT16)(offsetof(TPMT_SYM_DEF, mode))}},
693     // TPMT_SYM_DEF_OBJECT_DATA
694     {STRUCTURE_MTYPE, 3, {
695             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
696                 TPMI_ALG_SYM_OBJECT_MARSHAL_REF,
697                 (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, algorithm)),
698             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
699                 TPMU_SYM_KEY_BITS_MARSHAL_REF,
700                 (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, keyBits)),
701             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
702                 TPMU_SYM_MODE_MARSHAL_REF,
703                 (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, mode))}},
704     // TPM2B_SYM_KEY_DATA
705     {TPM2B_MTYPE, Type29_MARSHAL_REF},
706     // TPMS_SYMCIPHER_PARMS_DATA
707     {STRUCTURE_MTYPE, 1, {
708             SET_ELEMENT_TYPE(SIMPLE_STYPE),
709                 TPMT_SYM_DEF_OBJECT_MARSHAL_REF,
710                 (UINT16)(offsetof(TPMS_SYMCIPHER_PARMS, sym))}},
711     // TPM2B_LABEL_DATA
712     {TPM2B_MTYPE, Type30_MARSHAL_REF},
713     // TPMS_DERIVE_DATA
714     {STRUCTURE_MTYPE, 2, {
715             SET_ELEMENT_TYPE(SIMPLE_STYPE),
716                 TPM2B_LABEL_MARSHAL_REF,
717                 (UINT16)(offsetof(TPMS_DERIVE, label)),
718             SET_ELEMENT_TYPE(SIMPLE_STYPE),
719                 TPM2B_LABEL_MARSHAL_REF,
720                 (UINT16)(offsetof(TPMS_DERIVE, context))}},
721     // TPM2B_DERIVE_DATA
722     {TPM2B_MTYPE, Type31_MARSHAL_REF},
723     // TPM2B_SENSITIVE_DATA_DATA
724     {TPM2B_MTYPE, Type32_MARSHAL_REF},
725     // TPMS_SENSITIVE_CREATE_DATA
726     {STRUCTURE_MTYPE, 2, {
```

```
727             SET_ELEMENT_TYPE(SIMPLE_STYPE),
728                 TPM2B_AUTH_MARSHAL_REF,
729                 (UINT16)(offsetof(TPMS_SENSITIVE_CREATE, userAuth)),
730             SET_ELEMENT_TYPE(SIMPLE_STYPE),
731                 TPM2B_SENSITIVE_DATA_MARSHAL_REF,
732                 (UINT16)(offsetof(TPMS_SENSITIVE_CREATE, data))}},
733     // TPM2B_SENSITIVE_CREATE_DATA
734     {TPM2BS_MTYPE,
735         (UINT8)(offsetof(TPM2B_SENSITIVE_CREATE, sensitive))|SIZE_EQUAL,
736         UINT16_MARSHAL_REF,
737         TPMS_SENSITIVE_CREATE_MARSHAL_REF},
738     // TPMS_SCHEME_HASH_DATA
739     {STRUCTURE_MTYPE, 1, {
740             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
741                 TPMI_ALG_HASH_MARSHAL_REF,
742                 (UINT16)(offsetof(TPMS_SCHEME_HASH, hashAlg))}},
743     // TPMS_SCHEME_ECDAA_DATA
744     {STRUCTURE_MTYPE, 2, {
745             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
746                 TPMI_ALG_HASH_MARSHAL_REF,
747                 (UINT16)(offsetof(TPMS_SCHEME_ECDAA, hashAlg)),
748             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
749                 UINT16_MARSHAL_REF,
750                 (UINT16)(offsetof(TPMS_SCHEME_ECDAA, count))}},
751     // TPMI_ALG_KEYEDHASH_SCHEME_DATA
752     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
753         {TPM_ALG_NULL,
754         RANGE(5, 10, UINT16),
755         ((ALG_HMAC << 0)|(ALG_XOR << 5))}},
756     // TPMS_SCHEME_XOR_DATA
757     {STRUCTURE_MTYPE, 2, {
758             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
759                 TPMI_ALG_HASH_MARSHAL_REF,
760                 (UINT16)(offsetof(TPMS_SCHEME_XOR, hashAlg)),
761             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
762                 TPMI_ALG_KDF_MARSHAL_REF|NULL_FLAG,
763                 (UINT16)(offsetof(TPMS_SCHEME_XOR, kdf))}},
764     // TPMU_SCHEME_KEYEDHASH_DATA
765     {3, 0, (UINT16)(offsetof(TPMU_SCHEME_KEYEDHASH_mst, marshalingTypes)),
766         {(UINT32)TPM_ALG_HMAC, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
767         {(UINT16)(TPMS_SCHEME_HMAC_MARSHAL_REF),
768          (UINT16)(TPMS_SCHEME_XOR_MARSHAL_REF),
769          (UINT16)(UINT0_MARSHAL_REF)}
770     },
771     // TPMT_KEYEDHASH_SCHEME_DATA
772     {STRUCTURE_MTYPE, 2, {
773             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
774                 TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF,
775                 (UINT16)(offsetof(TPMT_KEYEDHASH_SCHEME, scheme)),
776             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
777                 TPMU_SCHEME_KEYEDHASH_MARSHAL_REF,
778                 (UINT16)(offsetof(TPMT_KEYEDHASH_SCHEME, details))}},
779     // TPMU_SIG_SCHEME_DATA
780     {8, 0, (UINT16)(offsetof(TPMU_SIG_SCHEME_mst, marshalingTypes)),
781         {(UINT32)TPM_ALG_ECDAA,     (UINT32)TPM_ALG_RSASSA,
782          (UINT32)TPM_ALG_RSAPSS,    (UINT32)TPM_ALG_ECDSA,
783          (UINT32)TPM_ALG_SM2,       (UINT32)TPM_ALG_ECSCHNORR,
784          (UINT32)TPM_ALG_HMAC,      (UINT32)TPM_ALG_NULL},
785         {(UINT16)(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF),
786          (UINT16)(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
787          (UINT16)(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
788          (UINT16)(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
789          (UINT16)(TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
790          (UINT16)(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
791          (UINT16)(TPMS_SCHEME_HMAC_MARSHAL_REF),
792          (UINT16)(UINT0_MARSHAL_REF)}
```

```
793     },
794     // TPMT_SIG_SCHEME_DATA
795     {STRUCTURE_MTYPE, 2, {
796             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
797                 TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
798                 (UINT16)(offsetof(TPMT_SIG_SCHEME, scheme)),
799             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
800                 TPMU_SIG_SCHEME_MARSHAL_REF,
801                 (UINT16)(offsetof(TPMT_SIG_SCHEME, details))}},
802     // TPMU_KDF_SCHEME_DATA
803     {5, 0, (UINT16)(offsetof(TPMU_KDF_SCHEME_mst, marshalingTypes)),
804         {(UINT32)TPM_ALG_MGF1,           (UINT32)TPM_ALG_KDF1_SP800_56A,
805          (UINT32)TPM_ALG_KDF2,           (UINT32)TPM_ALG_KDF1_SP800_108,
806          (UINT32)TPM_ALG_NULL},
807         {(UINT16)(TPMS_SCHEME_MGF1_MARSHAL_REF),
808          (UINT16)(TPMS_SCHEME_KDF1_SP800_56A_MARSHAL_REF),
809          (UINT16)(TPMS_SCHEME_KDF2_MARSHAL_REF),
810          (UINT16)(TPMS_SCHEME_KDF1_SP800_108_MARSHAL_REF),
811          (UINT16)(UINT0_MARSHAL_REF)}
812     },
813     // TPMT_KDF_SCHEME_DATA
814     {STRUCTURE_MTYPE, 2, {
815             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
816                 TPMI_ALG_KDF_MARSHAL_REF,
817                 (UINT16)(offsetof(TPMT_KDF_SCHEME, scheme)),
818             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
819                 TPMU_KDF_SCHEME_MARSHAL_REF,
820                 (UINT16)(offsetof(TPMT_KDF_SCHEME, details))}},
821     // TPMI_ALG_ASYM_SCHEME_DATA
822     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
823         {TPM_ALG_NULL,
824         RANGE(20, 29, UINT16),
825         ((ALG_RSASSA << 0)     | (ALG_RSAES << 1)      | (ALG_RSAPSS << 2)     |
826          (ALG_OAEP << 3)       | (ALG_ECDSA << 4)      | (ALG_ECDH << 5)       |
827          (ALG_ECDAA << 6)      | (ALG_SM2 << 7)        | (ALG_ECSCHNORR << 8) |
828          (ALG_ECMQV << 9))}},
829     // TPMU_ASYM_SCHEME_DATA
830     {11, 0, (UINT16)(offsetof(TPMU_ASYM_SCHEME_mst, marshalingTypes)),
831         {(UINT32)TPM_ALG_ECDH,       (UINT32)TPM_ALG_ECMQV,
832          (UINT32)TPM_ALG_ECDAA,      (UINT32)TPM_ALG_RSASSA,
833          (UINT32)TPM_ALG_RSAPSS,     (UINT32)TPM_ALG_ECDSA,
834          (UINT32)TPM_ALG_SM2,        (UINT32)TPM_ALG_ECSCHNORR,
835          (UINT32)TPM_ALG_RSAES,      (UINT32)TPM_ALG_OAEP,
836          (UINT32)TPM_ALG_NULL},
837         {(UINT16)(TPMS_KEY_SCHEME_ECDH_MARSHAL_REF),
838          (UINT16)(TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF),
839          (UINT16)(TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF),
840          (UINT16)(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
841          (UINT16)(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
842          (UINT16)(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
843          (UINT16)(TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
844          (UINT16)(TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
845          (UINT16)(TPMS_ENC_SCHEME_RSAES_MARSHAL_REF),
846          (UINT16)(TPMS_ENC_SCHEME_OAEP_MARSHAL_REF),
847          (UINT16)(UINT0_MARSHAL_REF)}
848     },
849     // TPMI_ALG_RSA_SCHEME_DATA
850     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
851         {TPM_ALG_NULL,
852         RANGE(20, 23, UINT16),
853         ((ALG_RSASSA << 0)|(ALG_RSAES << 1)|(ALG_RSAPSS << 2)|(ALG_OAEP << 3))}},
854     // TPMT_RSA_SCHEME_DATA
855     {STRUCTURE_MTYPE, 2, {
856             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
857                 TPMI_ALG_RSA_SCHEME_MARSHAL_REF,
858                 (UINT16)(offsetof(TPMT_RSA_SCHEME, scheme)),
```

```
859             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
860                 TPMU_ASYM_SCHEME_MARSHAL_REF,
861                 (UINT16)(offsetof(TPMT_RSA_SCHEME, details))}},
862     // TPMI_ALG_RSA_DECRYPT_DATA
863     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
864         {TPM_ALG_NULL,
865         RANGE(21, 23, UINT16),
866         ((ALG_RSAES << 0)|(ALG_OAEP << 2))}},
867     // TPMT_RSA_DECRYPT_DATA
868     {STRUCTURE_MTYPE, 2, {
869             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
870                 TPMI_ALG_RSA_DECRYPT_MARSHAL_REF,
871                 (UINT16)(offsetof(TPMT_RSA_DECRYPT, scheme)),
872             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
873                 TPMU_ASYM_SCHEME_MARSHAL_REF,
874                 (UINT16)(offsetof(TPMT_RSA_DECRYPT, details))}},
875     // TPM2B_PUBLIC_KEY_RSA_DATA
876     {TPM2B_MTYPE, Type33_MARSHAL_REF},
877     // TPMI_RSA_KEY_BITS_DATA
878     {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
879         {3072*RSA_3072, 1024*RSA_1024, 2048*RSA_2048}},
880     // TPM2B_PRIVATE_KEY_RSA_DATA
881     {TPM2B_MTYPE, Type34_MARSHAL_REF},
882     // TPM2B_ECC_PARAMETER_DATA
883     {TPM2B_MTYPE, Type35_MARSHAL_REF},
884     // TPMS_ECC_POINT_DATA
885     {STRUCTURE_MTYPE, 2, {
886             SET_ELEMENT_TYPE(SIMPLE_STYPE),
887                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
888                 (UINT16)(offsetof(TPMS_ECC_POINT, x)),
889             SET_ELEMENT_TYPE(SIMPLE_STYPE),
890                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
891                 (UINT16)(offsetof(TPMS_ECC_POINT, y))}},
892     // TPM2B_ECC_POINT_DATA
893     {TPM2BS_MTYPE,
894         (UINT8)(offsetof(TPM2B_ECC_POINT, point))|SIZE_EQUAL,
895         UINT16_MARSHAL_REF,
896         TPMS_ECC_POINT_MARSHAL_REF},
897     // TPMI_ALG_ECC_SCHEME_DATA
898     {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
899         {TPM_ALG_NULL,
900         RANGE(24, 29, UINT16),
901         ((ALG_ECDSA << 0)     | (ALG_ECDH << 1)      | (ALG_ECDAA << 2)     |
902          (ALG_SM2 << 3)       | (ALG_ECSCHNORR << 4) | (ALG_ECMQV << 5))}},
903     // TPMI_ECC_CURVE_DATA
904     {MIN_MAX_MTYPE, TWO_BYTES|HAS_BITS, (UINT8)TPM_RC_CURVE,
905         {RANGE(1, 32, UINT16),
906         ((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
907          (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15)  |
908          (ECC_BN_P638 << 16)  | (ECC_SM2_P256 << 31))}},
909     // TPMT_ECC_SCHEME_DATA
910     {STRUCTURE_MTYPE, 2, {
911             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
912                 TPMI_ALG_ECC_SCHEME_MARSHAL_REF,
913                 (UINT16)(offsetof(TPMT_ECC_SCHEME, scheme)),
914             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
915                 TPMU_ASYM_SCHEME_MARSHAL_REF,
916                 (UINT16)(offsetof(TPMT_ECC_SCHEME, details))}},
917     // TPMS_ALGORITHM_DETAIL_ECC_DATA
918     {STRUCTURE_MTYPE, 11, {
919             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
920                 TPM_ECC_CURVE_MARSHAL_REF,
921                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, curveID)),
922             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
923                 UINT16_MARSHAL_REF,
924                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, keySize)),
```

```
925             SET_ELEMENT_TYPE(SIMPLE_STYPE),
926                 TPMT_KDF_SCHEME_MARSHAL_REF|NULL_FLAG,
927                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, kdf)),
928             SET_ELEMENT_TYPE(SIMPLE_STYPE),
929                 TPMT_ECC_SCHEME_MARSHAL_REF|NULL_FLAG,
930                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, sign)),
931             SET_ELEMENT_TYPE(SIMPLE_STYPE),
932                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
933                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, p)),
934             SET_ELEMENT_TYPE(SIMPLE_STYPE),
935                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
936                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, a)),
937             SET_ELEMENT_TYPE(SIMPLE_STYPE),
938                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
939                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, b)),
940             SET_ELEMENT_TYPE(SIMPLE_STYPE),
941                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
942                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, gX)),
943             SET_ELEMENT_TYPE(SIMPLE_STYPE),
944                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
945                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, gY)),
946             SET_ELEMENT_TYPE(SIMPLE_STYPE),
947                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
948                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, n)),
949             SET_ELEMENT_TYPE(SIMPLE_STYPE),
950                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
951                 (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, h))}},
952     // TPMS_SIGNATURE_RSA_DATA
953     {STRUCTURE_MTYPE, 2, {
954             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
955                 TPMI_ALG_HASH_MARSHAL_REF,
956                 (UINT16)(offsetof(TPMS_SIGNATURE_RSA, hash)),
957             SET_ELEMENT_TYPE(SIMPLE_STYPE),
958                 TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF,
959                 (UINT16)(offsetof(TPMS_SIGNATURE_RSA, sig))}},
960     // TPMS_SIGNATURE_ECC_DATA
961     {STRUCTURE_MTYPE, 3, {
962             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
963                 TPMI_ALG_HASH_MARSHAL_REF,
964                 (UINT16)(offsetof(TPMS_SIGNATURE_ECC, hash)),
965             SET_ELEMENT_TYPE(SIMPLE_STYPE),
966                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
967                 (UINT16)(offsetof(TPMS_SIGNATURE_ECC, signatureR)),
968             SET_ELEMENT_TYPE(SIMPLE_STYPE),
969                 TPM2B_ECC_PARAMETER_MARSHAL_REF,
970                 (UINT16)(offsetof(TPMS_SIGNATURE_ECC, signatureS))}},
971     // TPMU_SIGNATURE_DATA
972     {8, 0, (UINT16)(offsetof(TPMU_SIGNATURE_mst, marshalingTypes)),
973         {(UINT32)TPM_ALG_ECDAA,     (UINT32)TPM_ALG_RSASSA,
974          (UINT32)TPM_ALG_RSAPSS,    (UINT32)TPM_ALG_ECDSA,
975          (UINT32)TPM_ALG_SM2,       (UINT32)TPM_ALG_ECSCHNORR,
976          (UINT32)TPM_ALG_HMAC,      (UINT32)TPM_ALG_NULL},
977         {(UINT16)(TPMS_SIGNATURE_ECDAA_MARSHAL_REF),
978          (UINT16)(TPMS_SIGNATURE_RSASSA_MARSHAL_REF),
979          (UINT16)(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF),
980          (UINT16)(TPMS_SIGNATURE_ECDSA_MARSHAL_REF),
981          (UINT16)(TPMS_SIGNATURE_SM2_MARSHAL_REF),
982          (UINT16)(TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF),
983          (UINT16)(TPMT_HA_MARSHAL_REF),
984          (UINT16)(UINT0_MARSHAL_REF)}
985     },
986     // TPMT_SIGNATURE_DATA
987     {STRUCTURE_MTYPE, 2, {
988             SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
989                 TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
990                 (UINT16)(offsetof(TPMT_SIGNATURE, sigAlg)),
```

```
991             SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
992                 TPMU_SIGNATURE_MARSHAL_REF,
993                 (UINT16)(offsetof(TPMT_SIGNATURE, signature))},
994     // TPMU_ENCRYPTED_SECRET_DATA
995     {4, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_ENCRYPTED_SECRET_mst, marshalingTypes)),
996         {(UINT32)TPM_ALG_ECC,       (UINT32)TPM_ALG_RSA,
997          (UINT32)TPM_ALG_SYMCIPHER, (UINT32)TPM_ALG_KEYEDHASH},
998         {(UINT16)(sizeof(TPMS_ECC_POINT)), (UINT16)(MAX_RSA_KEY_BYTES),
999          (UINT16)(sizeof(TPM2B_DIGEST)),   (UINT16)(sizeof(TPM2B_DIGEST))}
1000    },
1001    // TPM2B_ENCRYPTED_SECRET_DATA
1002    {TPM2B_MTYPE, Type36_MARSHAL_REF},
1003    // TPMI_ALG_PUBLIC_DATA
1004    {MIN_MAX_MTYPE, TWO_BYTES|HAS_BITS, (UINT8)TPM_RC_TYPE,
1005        {RANGE(1, 37, UINT16),
1006        ((ALG_RSA << 0)|(ALG_KEYEDHASH << 7)),
1007        ((ALG_ECC << 2)|(ALG_SYMCIPHER << 4))}},
1008    // TPMU_PUBLIC_ID_DATA
1009    {4, 0, (UINT16)(offsetof(TPMU_PUBLIC_ID_mst, marshalingTypes)),
1010        {(UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1011         (UINT32)TPM_ALG_RSA,       (UINT32)TPM_ALG_ECC},
1012        {(UINT16)(TPM2B_DIGEST_MARSHAL_REF),
1013         (UINT16)(TPM2B_DIGEST_MARSHAL_REF),
1014         (UINT16)(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF),
1015         (UINT16)(TPMS_ECC_POINT_MARSHAL_REF)}
1016    },
1017    // TPMS_KEYEDHASH_PARMS_DATA
1018    {STRUCTURE_MTYPE, 1, {
1019            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1020                TPMT_KEYEDHASH_SCHEME_MARSHAL_REF|NULL_FLAG,
1021                (UINT16)(offsetof(TPMS_KEYEDHASH_PARMS, scheme))}},
1022    // TPMS_RSA_PARMS_DATA
1023    {STRUCTURE_MTYPE, 4, {
1024            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1025                TPMT_SYM_DEF_OBJECT_MARSHAL_REF|NULL_FLAG,
1026                (UINT16)(offsetof(TPMS_RSA_PARMS, symmetric)),
1027            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1028                TPMT_RSA_SCHEME_MARSHAL_REF|NULL_FLAG,
1029                (UINT16)(offsetof(TPMS_RSA_PARMS, scheme)),
1030            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1031                TPMI_RSA_KEY_BITS_MARSHAL_REF,
1032                (UINT16)(offsetof(TPMS_RSA_PARMS, keyBits)),
1033            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1034                UINT32_MARSHAL_REF,
1035                (UINT16)(offsetof(TPMS_RSA_PARMS, exponent))}},
1036    // TPMS_ECC_PARMS_DATA
1037    {STRUCTURE_MTYPE, 4, {
1038            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1039                TPMT_SYM_DEF_OBJECT_MARSHAL_REF|NULL_FLAG,
1040                (UINT16)(offsetof(TPMS_ECC_PARMS, symmetric)),
1041            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1042                TPMT_ECC_SCHEME_MARSHAL_REF|NULL_FLAG,
1043                (UINT16)(offsetof(TPMS_ECC_PARMS, scheme)),
1044            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1045                TPMI_ECC_CURVE_MARSHAL_REF,
1046                (UINT16)(offsetof(TPMS_ECC_PARMS, curveID)),
1047            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1048                TPMT_KDF_SCHEME_MARSHAL_REF|NULL_FLAG,
1049                (UINT16)(offsetof(TPMS_ECC_PARMS, kdf))}},
1050    // TPMU_PUBLIC_PARMS_DATA
1051    {4, 0, (UINT16)(offsetof(TPMU_PUBLIC_PARMS_mst, marshalingTypes)),
1052        {(UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1053         (UINT32)TPM_ALG_RSA,       (UINT32)TPM_ALG_ECC},
1054        {(UINT16)(TPMS_KEYEDHASH_PARMS_MARSHAL_REF),
1055         (UINT16)(TPMS_SYMCIPHER_PARMS_MARSHAL_REF),
1056         (UINT16)(TPMS_RSA_PARMS_MARSHAL_REF),
```

```
1057        (UINT16)(TPMS_ECC_PARMS_MARSHAL_REF)}
1058    },
1059    // TPMT_PUBLIC_PARMS_DATA
1060    {STRUCTURE_MTYPE, 2, {
1061            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1062                TPMI_ALG_PUBLIC_MARSHAL_REF,
1063                (UINT16)(offsetof(TPMT_PUBLIC_PARMS, type)),
1064            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1065                TPMU_PUBLIC_PARMS_MARSHAL_REF,
1066                (UINT16)(offsetof(TPMT_PUBLIC_PARMS, parameters))}},
1067    // TPMT_PUBLIC_DATA
1068    {STRUCTURE_MTYPE, 6, {
1069            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1070                TPMI_ALG_PUBLIC_MARSHAL_REF,
1071                (UINT16)(offsetof(TPMT_PUBLIC, type)),
1072            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
1073                TPMI_ALG_HASH_MARSHAL_REF,
1074                (UINT16)(offsetof(TPMT_PUBLIC, nameAlg)),
1075            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1076                TPMA_OBJECT_MARSHAL_REF,
1077                (UINT16)(offsetof(TPMT_PUBLIC, objectAttributes)),
1078            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1079                TPM2B_DIGEST_MARSHAL_REF,
1080                (UINT16)(offsetof(TPMT_PUBLIC, authPolicy)),
1081            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1082                TPMU_PUBLIC_PARMS_MARSHAL_REF,
1083                (UINT16)(offsetof(TPMT_PUBLIC, parameters)),
1084            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1085                TPMU_PUBLIC_ID_MARSHAL_REF,
1086                (UINT16)(offsetof(TPMT_PUBLIC, unique))}},
1087    // TPM2B_PUBLIC_DATA
1088    {TPM2BS_MTYPE,
1089        (UINT8)(offsetof(TPM2B_PUBLIC, publicArea))|SIZE_EQUAL|ELEMENT_PROPAGATE,
1090        UINT16_MARSHAL_REF,
1091        TPMT_PUBLIC_MARSHAL_REF},
1092    // TPM2B_TEMPLATE_DATA
1093    {TPM2B_MTYPE, Type37_MARSHAL_REF},
1094    // TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA
1095    {TPM2B_MTYPE, Type38_MARSHAL_REF},
1096    // TPMU_SENSITIVE_COMPOSITE_DATA
1097    {4, 0, (UINT16)(offsetof(TPMU_SENSITIVE_COMPOSITE_mst, marshalingTypes)),
1098        {(UINT32)TPM_ALG_RSA,       (UINT32)TPM_ALG_ECC,
1099         (UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER},
1100        {(UINT16)(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF),
1101         (UINT16)(TPM2B_ECC_PARAMETER_MARSHAL_REF),
1102         (UINT16)(TPM2B_SENSITIVE_DATA_MARSHAL_REF),
1103         (UINT16)(TPM2B_SYM_KEY_MARSHAL_REF)}
1104    },
1105    // TPMT_SENSITIVE_DATA
1106    {STRUCTURE_MTYPE, 4, {
1107            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1108                TPMI_ALG_PUBLIC_MARSHAL_REF,
1109                (UINT16)(offsetof(TPMT_SENSITIVE, sensitiveType)),
1110            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1111                TPM2B_AUTH_MARSHAL_REF,
1112                (UINT16)(offsetof(TPMT_SENSITIVE, authValue)),
1113            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1114                TPM2B_DIGEST_MARSHAL_REF,
1115                (UINT16)(offsetof(TPMT_SENSITIVE, seedValue)),
1116            SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1117                TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF,
1118                (UINT16)(offsetof(TPMT_SENSITIVE, sensitive))}},
1119    // TPM2B_SENSITIVE_DATA
1120    {TPM2BS_MTYPE,
1121        (UINT8)(offsetof(TPM2B_SENSITIVE, sensitiveArea)),
1122        UINT16_MARSHAL_REF,
```

```
1123        TPMT_SENSITIVE_MARSHAL_REF},
1124    // TPM2B_PRIVATE_DATA
1125    {TPM2B_MTYPE, Type39_MARSHAL_REF},
1126    // TPM2B_ID_OBJECT_DATA
1127    {TPM2B_MTYPE, Type40_MARSHAL_REF},
1128    // TPMS_NV_PIN_COUNTER_PARAMETERS_DATA
1129    {STRUCTURE_MTYPE, 2, {
1130            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1131                UINT32_MARSHAL_REF,
1132                (UINT16)(offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinCount)),
1133            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1134                UINT32_MARSHAL_REF,
1135                (UINT16)(offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinLimit))}},
1136    // TPMA_NV_DATA
1137    {ATTRIBUTES_MTYPE, FOUR_BYTES, 0x01F00300},
1138    // TPMS_NV_PUBLIC_DATA
1139    {STRUCTURE_MTYPE, 5, {
1140            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1141                TPMI_RH_NV_INDEX_MARSHAL_REF,
1142                (UINT16)(offsetof(TPMS_NV_PUBLIC, nvIndex)),
1143            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1144                TPMI_ALG_HASH_MARSHAL_REF,
1145                (UINT16)(offsetof(TPMS_NV_PUBLIC, nameAlg)),
1146            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1147                TPMA_NV_MARSHAL_REF,
1148                (UINT16)(offsetof(TPMS_NV_PUBLIC, attributes)),
1149            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1150                TPM2B_DIGEST_MARSHAL_REF,
1151                (UINT16)(offsetof(TPMS_NV_PUBLIC, authPolicy)),
1152            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1153                Type41_MARSHAL_REF,
1154                (UINT16)(offsetof(TPMS_NV_PUBLIC, dataSize))}},
1155    // TPM2B_NV_PUBLIC_DATA
1156    {TPM2BS_MTYPE,
1157        (UINT8)(offsetof(TPM2B_NV_PUBLIC, nvPublic))|SIZE_EQUAL,
1158        UINT16_MARSHAL_REF,
1159        TPMS_NV_PUBLIC_MARSHAL_REF},
1160    // TPM2B_CONTEXT_SENSITIVE_DATA
1161    {TPM2B_MTYPE, Type42_MARSHAL_REF},
1162    // TPMS_CONTEXT_DATA_DATA
1163    {STRUCTURE_MTYPE, 2, {
1164            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1165                TPM2B_DIGEST_MARSHAL_REF,
1166                (UINT16)(offsetof(TPMS_CONTEXT_DATA, integrity)),
1167            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1168                TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF,
1169                (UINT16)(offsetof(TPMS_CONTEXT_DATA, encrypted))}},
1170    // TPM2B_CONTEXT_DATA_DATA
1171    {TPM2B_MTYPE, Type43_MARSHAL_REF},
1172    // TPMS_CONTEXT_DATA
1173    {STRUCTURE_MTYPE, 4, {
1174            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
1175                UINT64_MARSHAL_REF,
1176                (UINT16)(offsetof(TPMS_CONTEXT, sequence)),
1177            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1178                TPMI_DH_SAVED_MARSHAL_REF,
1179                (UINT16)(offsetof(TPMS_CONTEXT, savedHandle)),
1180            SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1181                TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
1182                (UINT16)(offsetof(TPMS_CONTEXT, hierarchy)),
1183            SET_ELEMENT_TYPE(SIMPLE_STYPE),
1184                TPM2B_CONTEXT_DATA_MARSHAL_REF,
1185                (UINT16)(offsetof(TPMS_CONTEXT, contextBlob))}},
1186    // TPMS_CREATION_DATA_DATA
1187    {STRUCTURE_MTYPE, 7, {
1188            SET_ELEMENT_TYPE(SIMPLE_STYPE),
```

```
            TPML_PCR_SELECTION_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, pcrSelect)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DIGEST_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, pcrDigest)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(ONE_BYTES),
            TPMA_LOCALITY_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, locality)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
            TPM_ALG_ID_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, parentNameAlg)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_NAME_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, parentName)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_NAME_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, parentQualifiedName)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE),
            TPM2B_DATA_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_CREATION_DATA, outsideInfo))}},
// TPM2B_CREATION_DATA_DATA
{TPM2BS_MTYPE,
    (UINT8)(offsetof(TPM2B_CREATION_DATA, creationData))|SIZE_EQUAL,
    UINT16_MARSHAL_REF,
    TPMS_CREATION_DATA_MARSHAL_REF},
// TPM_AT_DATA
{TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4,
    {TPM_AT_ANY, TPM_AT_ERROR, TPM_AT_PV1, TPM_AT_VEND}},
// TPMS_AC_OUTPUT_DATA
{STRUCTURE_MTYPE, 2, {
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
            TPM_AT_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_AC_OUTPUT, tag)),
        SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
            UINT32_MARSHAL_REF,
            (UINT16)(offsetof(TPMS_AC_OUTPUT, data))}},
// TPML_AC_CAPABILITIES_DATA
{LIST_MTYPE,
    (UINT8)(offsetof(TPML_AC_CAPABILITIES, acCapabilities)),
    Type44_MARSHAL_REF,
    TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX},
// Type00_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, sizeof(TPMU_HA), UINT16)}},
// Type01_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, sizeof(TPMT_HA), UINT16)}},
// Type02_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, 1024, UINT16)}},
// Type03_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, MAX_DIGEST_BUFFER, UINT16)}},
// Type04_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, MAX_NV_BUFFER_SIZE, UINT16)}},
// Type05_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, sizeof(UINT64), UINT16)}},
// Type06_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, MAX_SYM_BLOCK_SIZE, UINT16)}},
// Type07_DATA
{MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
    {RANGE(0, sizeof(TPMU_NAME), UINT16)}},
// Type08_DATA
```

```
1255    {MIN_MAX_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE,
1256        {RANGE(PCR_SELECT_MIN, PCR_SELECT_MAX, UINT8)}},
1257    // Type10_DATA
1258    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1259        {TPM_ST_CREATION}},
1260    // Type11_DATA
1261    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1262        {TPM_ST_VERIFIED}},
1263    // Type12_DATA
1264    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 2,
1265        {TPM_ST_AUTH_SECRET, TPM_ST_AUTH_SIGNED}},
1266    // Type13_DATA
1267    {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1268        {TPM_ST_HASHCHECK}},
1269    // Type15_DATA
1270    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1271        {RANGE(0, MAX_CAP_CC, UINT32)}},
1272    // Type17_DATA
1273    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1274        {RANGE(0, MAX_ALG_LIST_SIZE, UINT32)}},
1275    // Type18_DATA
1276    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1277        {RANGE(0, MAX_CAP_HANDLES, UINT32)}},
1278    // Type19_DATA
1279    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1280        {RANGE(2, 8, UINT32)}},
1281    // Type20_DATA
1282    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1283        {RANGE(0, HASH_COUNT, UINT32)}},
1284    // Type22_DATA
1285    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1286        {RANGE(0, MAX_CAP_ALGS, UINT32)}},
1287    // Type23_DATA
1288    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1289        {RANGE(0, MAX_TPM_PROPERTIES, UINT32)}},
1290    // Type24_DATA
1291    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1292        {RANGE(0, MAX_PCR_PROPERTIES, UINT32)}},
1293    // Type25_DATA
1294    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1295        {RANGE(0, MAX_ECC_CURVES, UINT32)}},
1296    // Type26_DATA
1297    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1298        {RANGE(0, MAX_TAGGED_POLICIES, UINT32)}},
1299    // Type27_DATA
1300    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1301        {RANGE(0, MAX_ACT_DATA, UINT32)}},
1302    // Type28_DATA
1303    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1304        {RANGE(0, sizeof(TPMS_ATTEST), UINT16)}},
1305    // Type29_DATA
1306    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1307        {RANGE(0, MAX_SYM_KEY_BYTES, UINT16)}},
1308    // Type30_DATA
1309    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1310        {RANGE(0, LABEL_MAX_BUFFER, UINT16)}},
1311    // Type31_DATA
1312    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1313        {RANGE(0, sizeof(TPMS_DERIVE), UINT16)}},
1314    // Type32_DATA
1315    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1316        {RANGE(0, sizeof(TPMU_SENSITIVE_CREATE), UINT16)}},
1317    // Type33_DATA
1318    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1319        {RANGE(0, MAX_RSA_KEY_BYTES, UINT16)}},
1320    // Type34_DATA
```

```
1321    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1322        {RANGE(0, RSA_PRIVATE_SIZE, UINT16)}},
1323    // Type35_DATA
1324    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1325        {RANGE(0, MAX_ECC_KEY_BYTES, UINT16)}},
1326    // Type36_DATA
1327    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1328        {RANGE(0, sizeof(TPMU_ENCRYPTED_SECRET), UINT16)}},
1329    // Type37_DATA
1330    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1331        {RANGE(0, sizeof(TPMT_PUBLIC), UINT16)}},
1332    // Type38_DATA
1333    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1334        {RANGE(0, PRIVATE_VENDOR_SPECIFIC_BYTES, UINT16)}},
1335    // Type39_DATA
1336    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1337        {RANGE(0, sizeof(_PRIVATE), UINT16)}},
1338    // Type40_DATA
1339    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1340        {RANGE(0, sizeof(TPMS_ID_OBJECT), UINT16)}},
1341    // Type41_DATA
1342    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1343        {RANGE(0, MAX_NV_INDEX_SIZE, UINT16)}},
1344    // Type42_DATA
1345    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1346        {RANGE(0, MAX_CONTEXT_SIZE, UINT16)}},
1347    // Type43_DATA
1348    {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1349        {RANGE(0, sizeof(TPMS_CONTEXT_DATA), UINT16)}},
1350    // Type44_DATA
1351    {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1352        {RANGE(0, MAX_AC_CAPABILITIES, UINT32)}}
1353    };
1354    #endif // TABLE_DRIVEN_MARSHAL
```

9.11 MathOnByteBuffers.c

9.11.1 Introduction

This file contains implementation of the math functions that are performed with canonical integers in byte buffers. The canonical integer is big-endian bytes.

```
1   #include "Tpm.h"
```

9.11.2 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)

```
2   LIB_EXPORT int
3   UnsignedCompareB(
4       UINT32           aSize,         // IN: size of a
5       const BYTE      *a,             // IN: a
6       UINT32           bSize,         // IN: size of b
7       const BYTE      *b              // IN: b
8       )
9   {
10      UINT32              i;
11      if(aSize > bSize)
12          return 1;
13      else if(aSize < bSize)
14          return -1;
15      else
16      {
17          for(i = 0; i < aSize; i++)
18          {
19              if(a[i] != b[i])
20                  return (a[i] > b[i]) ? 1 : -1;
21          }
22      }
23      return 0;
24  }
```

9.11.2.2 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     }
```

9.12 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.

```
void
MemoryCopy(
    void        *dest,
    const void  *src,
    int          sSize
    )
{
    if(dest != src)
        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--)
```

```
27          equal |= (*b1++ ^ *b2++);
28      return (equal == 0);
29  }
```

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.

```
30  LIB_EXPORT INT16
31  MemoryCopy2B(
32      TPM2B           *dest,          // OUT: receiving TPM2B
33      const TPM2B     *source,        // IN: source TPM2B
34      unsigned int     dSize          // IN: size of the receiving buffer
35      )
36  {
37      pAssert(dest != NULL);
38      if(source == NULL)
39          dest->size = 0;
40      else
41      {
42          pAssert(source->size <= dSize);
43          MemoryCopy(dest->buffer, source->buffer, source->size);
44          dest->size = source->size;
45      }
46      return dest->size;
47  }
```

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*)).

```
48  void
49  MemoryConcat2B(
50      TPM2B           *aInOut,        // IN/OUT: destination 2B
51      TPM2B           *bIn,           // IN: second 2B
52      unsigned int     aMaxSize       // IN: The size of aInOut.buffer (max values for
53                                      //     aInOut.size)
54      )
55  {
56      pAssert(bIn->size <= aMaxSize - aInOut->size);
57      MemoryCopy(&aInOut->buffer[aInOut->size], &bIn->buffer, bIn->size);
58      aInOut->size = aInOut->size + bIn->size;
59      return;
60  }
```

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

```
61  BOOL
62  MemoryEqual2B(
```

```
63      const TPM2B     *aIn,           // IN: compare value
64      const TPM2B     *bIn            // IN: compare value
65      )
66  {
67      if(aIn->size != bIn->size)
68          return FALSE;
69      return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
70  }
```

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.

```
71  void
72  MemorySet(
73      void            *dest,
74      int              value,
75      size_t           size
76      )
77  {
78      memset(dest, value, size);
79  }
```

9.12.3.7 MemoryPad2B()

Function to pad a TPM2B with zeros and adjust the size.

```
80  void
81  MemoryPad2B(
82      TPM2B           *b,
83      UINT16           newSize
84      )
85  {
86      MemorySet(&b->buffer[b->size], 0, newSize - b->size);
87      b->size = newSize;
88  }
```

9.12.3.8 Uint16ToByteArray()

Function to write an integer to a byte array

```
89  void
90  Uint16ToByteArray(
91      UINT16              i,
92      BYTE                *a
93      )
94  {
95      a[1] = (BYTE)(i); i >>= 8;
96      a[0] = (BYTE)(i);
97  }
```

9.12.3.9 Uint32ToByteArray()

Function to write an integer to a byte array

```
98  void
```

```
 99  Uint32ToByteArray(
100      UINT32              i,
101      BYTE                *a
102      )
103  {
104      a[3] = (BYTE)(i); i >>= 8;
105      a[2] = (BYTE)(i); i >>= 8;
106      a[1] = (BYTE)(i); i >>= 8;
107      a[0] = (BYTE)(i);
108  }
```

9.12.3.10 Uint64ToByteArray()

Function to write an integer to a byte array

```
109  void
110  Uint64ToByteArray(
111      UINT64               i,
112      BYTE                *a
113      )
114  {
115      a[7] = (BYTE)(i); i >>= 8;
116      a[6] = (BYTE)(i); i >>= 8;
117      a[5] = (BYTE)(i); i >>= 8;
118      a[4] = (BYTE)(i); i >>= 8;
119      a[3] = (BYTE)(i); i >>= 8;
120      a[2] = (BYTE)(i); i >>= 8;
121      a[1] = (BYTE)(i); i >>= 8;
122      a[0] = (BYTE)(i);
123  }
```

9.12.3.11 ByteArrayToUint8()

Function to write a UINT8 to a byte array. This is included for completeness and to allow certain macro expansions

```
124  UINT8
125  ByteArrayToUint8(
126      BYTE                *a
127  )
128  {
129      return          *a;
130  }
```

9.12.3.12 ByteArrayToUint16()

Function to write an integer to a byte array

```
131  UINT16
132  ByteArrayToUint16(
133      BYTE                *a
134  )
135  {
136      return ((UINT16)a[0] << 8) + a[1];
137  }
```

9.12.3.13 ByteArrayToUint32()

Function to write an integer to a byte array

```
138 UINT32
139 ByteArrayToUint32(
140     BYTE                *a
141 )
142 {
143     return (UINT32)((((((UINT32)a[0] << 8) + a[1]) << 8) + (UINT32)a[2]) << 8) + a[3];
144 }
```

9.12.3.14 ByteArrayToUint64()

Function to write an integer to a byte array

```
145 UINT64
146 ByteArrayToUint64(
147     BYTE                *a
148     )
149 {
150     return (((UINT64)BYTE_ARRAY_TO_UINT32(a)) << 32) + BYTE_ARRAY_TO_UINT32(&a[4]);
151 }
```

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).

```
BOOL
TPMRegisterStartup(
    void
    )
{
    g_initialized = TRUE;
    return TRUE;
}
```

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

```
#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

```
static BOOL
TPMPropertyIsDefined(
    TPM_PT           property,      // IN: property
    UINT32          *value          // OUT: property value
    )
{
    switch(property)
    {
        case TPM_PT_FAMILY_INDICATOR:
            // from the title page of the specification
            // For this specification, the value is "2.0".
            *value = TPM_SPEC_FAMILY;
            break;
        case TPM_PT_LEVEL:
            // from the title page of the specification
            *value = TPM_SPEC_LEVEL;
            break;
        case TPM_PT_REVISION:
            // from the title page of the specification
            *value = TPM_SPEC_VERSION;
            break;
        case TPM_PT_DAY_OF_YEAR:
            // computed from the date value on the title page of the specification
            *value = TPM_SPEC_DAY_OF_YEAR;
            break;
        case TPM_PT_YEAR:
            // from the title page of the specification
            *value = TPM_SPEC_YEAR;
            break;
        case TPM_PT_MANUFACTURER:
            // vendor ID unique to each TPM manufacturer
            *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
            break;
        case TPM_PT_VENDOR_STRING_1:
            // first four characters of the vendor ID string
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
```

```
38              break;
39          case TPM_PT_VENDOR_STRING_2:
40              // second four characters of the vendor ID string
41  #ifdef VENDOR_STRING_2
42              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
43  #else
44              *value = 0;
45  #endif
46              break;
47          case TPM_PT_VENDOR_STRING_3:
48              // third four characters of the vendor ID string
49  #ifdef VENDOR_STRING_3
50              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
51  #else
52              *value = 0;
53  #endif
54              break;
55          case TPM_PT_VENDOR_STRING_4:
56              // fourth four characters of the vendor ID string
57  #ifdef VENDOR_STRING_4
58              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
59  #else
60              *value = 0;
61  #endif
62              break;
63          case TPM_PT_VENDOR_TPM_TYPE:
64              // vendor-defined value indicating the TPM model
65              *value = 1;
66              break;
67          case TPM_PT_FIRMWARE_VERSION_1:
68              // more significant 32-bits of a vendor-specific value
69              *value = gp.firmwareV1;
70              break;
71          case TPM_PT_FIRMWARE_VERSION_2:
72              // less significant 32-bits of a vendor-specific value
73              *value = gp.firmwareV2;
74              break;
75          case TPM_PT_INPUT_BUFFER:
76              // maximum size of TPM2B_MAX_BUFFER
77              *value = MAX_DIGEST_BUFFER;
78              break;
79          case TPM_PT_HR_TRANSIENT_MIN:
80              // minimum number of transient objects that can be held in TPM
81              // RAM
82              *value = MAX_LOADED_OBJECTS;
83              break;
84          case TPM_PT_HR_PERSISTENT_MIN:
85              // minimum number of persistent objects that can be held in
86              // TPM NV memory
87              // In this implementation, there is no minimum number of
88              // persistent objects.
89              *value = MIN_EVICT_OBJECTS;
90              break;
91          case TPM_PT_HR_LOADED_MIN:
92              // minimum number of authorization sessions that can be held in
93              // TPM RAM
94              *value = MAX_LOADED_SESSIONS;
95              break;
96          case TPM_PT_ACTIVE_SESSIONS_MAX:
97              // number of authorization sessions that may be active at a time
98              *value = MAX_ACTIVE_SESSIONS;
99              break;
100         case TPM_PT_PCR_COUNT:
101             // number of PCR implemented
102             *value = IMPLEMENTATION_PCR;
103             break;
```

```
104             case TPM_PT_PCR_SELECT_MIN:
105                 // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106                 *value = PCR_SELECT_MIN;
107                 break;
108             case TPM_PT_CONTEXT_GAP_MAX:
109                 // maximum allowed difference (unsigned) between the contextID
110                 // values of two saved session contexts
111                 *value = ((UINT32)1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
112                 break;
113             case TPM_PT_NV_COUNTERS_MAX:
114                 // maximum number of NV indexes that are allowed to have the
115                 // TPMA_NV_COUNTER attribute SET
116                 // In this implementation, there is no limitation on the number
117                 // of counters, except for the size of the NV Index memory.
118                 *value = 0;
119                 break;
120             case TPM_PT_NV_INDEX_MAX:
121                 // maximum size of an NV index data area
122                 *value = MAX_NV_INDEX_SIZE;
123                 break;
124             case TPM_PT_MEMORY:
125                 // a TPMA_MEMORY indicating the memory management method for the TPM
126                 {
127                     union
128                     {
129                         TPMA_MEMORY     att;
130                         UINT32          u32;
131                     } attributes = { TPMA_ZERO_INITIALIZER() };
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 = { TPMA_ZERO_INITIALIZER() };
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 = { TPMA_ZERO_INITIALIZER() };
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

```
1   #include "Tpm.h"
```

9.16.3 RcSafeAddToResult()

Adds a modifier to a response code as long as the response code allows a modifier and no modifier has already been added.

```
2   TPM_RC
3   RcSafeAddToResult(
4       TPM_RC          responseCode,
5       TPM_RC          modifier
6       )
7   {
8       if((responseCode & RC_FMT1) && !(responseCode & 0xf40))
9          return responseCode + modifier;
10      else
11          return responseCode;
12  }
```

9.17 TpmFail.c

9.17.1 Includes, Defines, and Types

```
#define     TPM_FAIL_C
#include    "Tpm.h"
#include    <assert.h>
```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TpmTypes.h include. This will avoid a lot of alignment warnings from the compiler for the unaligned structures. The alignment of the structures is not important as this function does not use any of the structures in TpmTypes.h and only include it for the #defines of the capabilities, properties, and command code values.

```
#include "TpmTypes.h"
```

9.17.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```
typedef struct
{
    TPM_ST          tag;
    UINT32          size;
    TPM_RC          code;
} HEADER;
typedef struct
{
    BYTE            tag[sizeof(TPM_ST)];
    BYTE            size[sizeof(UINT32)];
    BYTE            code[sizeof(TPM_RC)];
} PACKED_HEADER;
typedef struct
{
    BYTE             size[sizeof(UINT16)];
    struct
    {
        BYTE         function[sizeof(UINT32)];
        BYTE         line[sizeof(UINT32)];
        BYTE         code[sizeof(UINT32)];
    } values;
    BYTE             returnCode[sizeof(TPM_RC)];
} GET_TEST_RESULT_PARAMETERS;
typedef struct
{
    BYTE         moreData[sizeof(TPMI_YES_NO)];
    BYTE         capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
    BYTE         tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
} GET_CAPABILITY_PARAMETERS;
typedef struct
{
    BYTE         header[sizeof(PACKED_HEADER)];
    BYTE         getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
} TEST_RESPONSE;
typedef struct
{
    BYTE         header[sizeof(PACKED_HEADER)];
    BYTE         getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
} CAPABILITY_RESPONSE;
typedef union
{
```

```
46      BYTE          test[sizeof(TEST_RESPONSE)];
47      BYTE          cap[sizeof(CAPABILITY_RESPONSE)];
48  } RESPONSES;
```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

NOTE: This is not in Global.c because of the specialized data definitions above. Since the data contained in this structure is not relevant outside of the execution of a single command (when the TPM is in failure mode. There is no compelling reason to move all the typedefs to Global.h and this structure to Global.c.

```
49  #ifndef __IGNORE_STATE__ // Don't define this value
50  static BYTE response[sizeof(RESPONSES)];
51  #endif
```

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
52  static INT32
53  MarshalUint16(
54      UINT16          integer,
55      BYTE            **buffer
56      )
57  {
58      UINT16_TO_BYTE_ARRAY(integer, *buffer);
59      *buffer += 2;
60      return 2;
61  }
```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
62  static INT32
63  MarshalUint32(
64      UINT32           integer,
65      BYTE            **buffer
66      )
67  {
68      UINT32_TO_BYTE_ARRAY(integer, *buffer);
69      *buffer += 4;
70      return 4;
71  }
```

9.17.3.3 Unmarshal32()

```
72  static BOOL Unmarshal32(
73      UINT32          *target,
74      BYTE            **buffer,
75      INT32           *size
76      )
77  {
78      if((*size -= 4) < 0)
79          return FALSE;
80      *target = BYTE_ARRAY_TO_UINT32(*buffer);
81      *buffer += 4;
82      return TRUE;
```

```
}
```

9.17.3.4 Unmarshal16()

```
static BOOL Unmarshal16(
    UINT16          *target,
    BYTE           **buffer,
    INT32           *size
)
{
    if((*size -= 2) < 0)
        return FALSE;
    *target = BYTE_ARRAY_TO_UINT16(*buffer);
    *buffer += 2;
    return TRUE;
}
```

9.17.4 Public Functions

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
#if SIMULATION
LIB_EXPORT void
SetForceFailureMode(
    void
    )
{
    g_forceFailureMode = TRUE;
    return;
}
#endif
```

9.17.4.2 TpmLogFailure()

This function saves the failure values when the code will continue to operate. It if similar to TpmFail() but returns to the caller. The assumption is that the caller will propagate a failure back up the stack.

```
void
TpmLogFailure(
#if FAIL_TRACE
    const char      *function,
    int              line,
#endif
    int              code
)
{
    // Save the values that indicate where the error occurred.
    // On a 64-bit machine, this may truncate the address of the string
    // of the function name where the error occurred.
#if FAIL_TRACE
    s_failFunction = (UINT32)(ptrdiff_t)function;
    s_failLine = line;
#else
    s_failFunction = 0;
    s_failLine = 0;
#endif
    s_failCode = code;

    // We are in failure mode
```

```
128         g_inFailureMode = TRUE;
129
130         return;
131     }
```

9.17.4.3 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().

```
132     NORETURN void
133     TpmFail(
134     #if FAIL_TRACE
135         const char      *function,
136         int              line,
137     #endif
138         int              code
139         )
140     {
141         // Save the values that indicate where the error occurred.
142         // On a 64-bit machine, this may truncate the address of the string
143         // of the function name where the error occurred.
144     #if FAIL_TRACE
145         s_failFunction = (UINT32)(ptrdiff_t)function;
146         s_failLine = line;
147     #else
148         s_failFunction = (UINT32)(ptrdiff_t)NULL;
149         s_failLine = 0;
150     #endif
151         s_failCode = code;
152
153         // We are in failure mode
154         g_inFailureMode = TRUE;
155
156         // if asserts are enabled, then do an assert unless the failure mode code
157         // is being tested.
158     #if SIMULATION
159     #   ifndef NDEBUG
160         assert(g_forceFailureMode);
161     #   endif
162         // Clear this flag
163         g_forceFailureMode = FALSE;
164     #endif
165         // Jump to the failure mode code.
166         // Note: only get here if asserts are off or if we are testing failure mode
167         _plat__Fail();
168     }
```

9.17.4.4 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
169     void
170     TpmFailureMode(
171         unsigned int     inRequestSize,     // IN: command buffer size
172         unsigned char   *inRequest,         // IN: command buffer
173         unsigned int    *outResponseSize,   // OUT: response buffer size
174         unsigned char   **outResponse       // OUT: response buffer
175         )
176     {
177         UINT32           marshalSize;
178         UINT32           capability;
179         HEADER           header;    // unmarshaled command header
```

```
180         UINT32           pt;    // unmarshaled property type
181         UINT32           count; // unmarshaled property count
182         UINT8           *buffer = inRequest;
183         INT32            size = inRequestSize;
184
185         // If there is no command buffer, then just return TPM_RC_FAILURE
186         if(inRequestSize == 0 || inRequest == NULL)
187             goto FailureModeReturn;
188         // If the header is not correct for TPM2_GetCapability() or
189         // TPM2_GetTestResult() then just return the in failure mode response;
190         if(! (Unmarshal16(&header.tag,  &buffer, &size)
191            && Unmarshal32(&header.size, &buffer, &size)
192            && Unmarshal32(&header.code, &buffer, &size)))
193             goto FailureModeReturn;
194         if(header.tag != TPM_ST_NO_SESSIONS
195            || header.size < 10)
196             goto FailureModeReturn;
197         switch(header.code)
198         {
199             case TPM_CC_GetTestResult:
200                 // make sure that the command size is correct
201                 if(header.size != 10)
202                     goto FailureModeReturn;
203                 buffer = &response[10];
204                 marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
205                 marshalSize += MarshalUint32(s_failFunction, &buffer);
206                 marshalSize += MarshalUint32(s_failLine, &buffer);
207                 marshalSize += MarshalUint32(s_failCode, &buffer);
208                 if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
209                     marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
210                 else
211                     marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
212                 break;
213             case TPM_CC_GetCapability:
214                 // make sure that the size of the command is exactly the size
215                 // returned for the capability, property, and count
216                 if(header.size != (10 + (3 * sizeof(UINT32)))
217                         // also verify that this is requesting TPM properties
218                    || !Unmarshal32(&capability, &buffer, &size)
219                    || capability != TPM_CAP_TPM_PROPERTIES
220                    || !Unmarshal32(&pt, &buffer, &size)
221                    || !Unmarshal32(&count, &buffer, &size))
222                     goto FailureModeReturn;
223                 // If in failure mode because of an unrecoverable read error, and the
224                 // property is 0 and the count is 0, then this is an indication to
225                 // re-manufacture the TPM. Do the re-manufacture but stay in failure
226                 // mode until the TPM is reset.
227                 // Note: this behavior is not required by the specification and it is
228                 // OK to leave the TPM permanently bricked due to an unrecoverable NV
229                 // error.
230                 if(count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
231                 {
232                     g_manufactured = FALSE;
233                     TPM_Manufacture(0);
234                 }
235                 if(count > 0)
236                     count = 1;
237                 else if(pt > TPM_PT_FIRMWARE_VERSION_2)
238                     count = 0;
239                 if(pt < TPM_PT_MANUFACTURER)
240                     pt = TPM_PT_MANUFACTURER;
241                 // set up for return
242                 buffer = &response[10];
243                 // if the request was for a PT less than the last one
244                 // then we indicate more, otherwise, not.
245                 if(pt < TPM_PT_FIRMWARE_VERSION_2)
```

```
246                 *buffer++ = YES;
247             else
248                 *buffer++ = NO;
249             marshalSize = 1;
250 
251             // indicate the capability type
252             marshalSize += MarshalUint32(capability, &buffer);
253             // indicate the number of values that are being returned (0 or 1)
254             marshalSize += MarshalUint32(count, &buffer);
255             // indicate the property
256             marshalSize += MarshalUint32(pt, &buffer);
257 
258             if(count > 0)
259                 switch(pt)
260                 {
261                     case TPM_PT_MANUFACTURER:
262                     // the vendor ID unique to each TPM manufacturer
263 #ifdef  MANUFACTURER
264                         pt = *(UINT32*)MANUFACTURER;
265 #else
266                         pt = 0;
267 #endif
268                         break;
269                     case TPM_PT_VENDOR_STRING_1:
270                         // the first four characters of the vendor ID string
271 #ifdef  VENDOR_STRING_1
272                         pt = *(UINT32*)VENDOR_STRING_1;
273 #else
274                         pt = 0;
275 #endif
276                         break;
277                     case TPM_PT_VENDOR_STRING_2:
278                         // the second four characters of the vendor ID string
279 #ifdef  VENDOR_STRING_2
280                         pt = *(UINT32*)VENDOR_STRING_2;
281 #else
282                         pt = 0;
283 #endif
284                         break;
285                     case TPM_PT_VENDOR_STRING_3:
286                         // the third four characters of the vendor ID string
287 #ifdef  VENDOR_STRING_3
288                         pt = *(UINT32*)VENDOR_STRING_3;
289 #else
290                         pt = 0;
291 #endif
292                         break;
293                     case TPM_PT_VENDOR_STRING_4:
294                         // the fourth four characters of the vendor ID string
295 #ifdef  VENDOR_STRING_4
296                         pt = *(UINT32*)VENDOR_STRING_4;
297 #else
298                         pt = 0;
299 #endif
300                         break;
301                     case TPM_PT_VENDOR_TPM_TYPE:
302                         // vendor-defined value indicating the TPM model
303                         // We just make up a number here
304                         pt = 1;
305                         break;
306                     case TPM_PT_FIRMWARE_VERSION_1:
307                         // the more significant 32-bits of a vendor-specific value
308                         // indicating the version of the firmware
309 #ifdef  FIRMWARE_V1
310                         pt = FIRMWARE_V1;
311 #else
```

```
312                         pt = 0;
313 #endif
314                         break;
315                     default: // TPM_PT_FIRMWARE_VERSION_2:
316                         // the less significant 32-bits of a vendor-specific value
317                         // indicating the version of the firmware
318 #ifdef  FIRMWARE_V2
319                         pt = FIRMWARE_V2;
320 #else
321                         pt = 0;
322 #endif
323                         break;
324                 }
325             marshalSize += MarshalUint32(pt, &buffer);
326             break;
327         default: // default for switch (cc)
328             goto FailureModeReturn;
329     }
330     // Now do the header
331     buffer = response;
332     marshalSize = marshalSize + 10; // Add the header size to the
333                                     // stuff already marshaled
334     MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
335     MarshalUint32(marshalSize, &buffer);  // responseSize
336     MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
337
338     *outResponseSize = marshalSize;
339     *outResponse = (unsigned char *)&response;
340     return;
341 FailureModeReturn:
342     buffer = response;
343     marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
344     marshalSize += MarshalUint32(10, &buffer);
345     marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
346     *outResponseSize = marshalSize;
347     *outResponse = (unsigned char *)response;
348     return;
349 }
```

9.17.4.5 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...

```
350 void
351 UnmarshalFail(
352     void            *type,
353     BYTE            **buffer,
354     INT32           *size
355     )
356 {
357     NOT_REFERENCED(type);
358     NOT_REFERENCED(buffer);
359     NOT_REFERENCED(size);
360     FAIL(FATAL_ERROR_INTERNAL);
361 }
```

10 Cryptographic Functions

10.1 Headers

10.1.1 BnValues.h

10.1.1.1 Introduction

This file contains the definitions needed for defining the internal BIGNUM structure. A BIGNUM is a pointer to a structure. The structure has three fields. The last field is and array (d) of crypt_uword_t. Each word is in machine format (big- or little-endian) with the words in ascending significance (i.e. words in little-endian order). This is the order that seems to be used in every big number library in the worlds, so...

The first field in the structure (allocated) is the number of words in *d*. This is the upper limit on the size of the number that can be held in the structure. This differs from libraries like OpenSSL as this is not intended to deal with numbers of arbitrary size; just numbers that are needed to deal with the algorithms that are defined in the TPM implementation.

The second field in the structure (size) is the number of significant words in *n*. When this number is zero, the number is zero. The word at used-1 should never be zero. All words between d[size] and d[allocated-1] should be zero.

10.1.1.2 Defines

```
#ifndef _BN_NUMBERS_H
#define _BN_NUMBERS_H
#if RADIX_BITS == 64
# define RADIX_LOG2         6
#elif RADIX_BITS == 32
#define RADIX_LOG2          5
#else
# error "Unsupported radix"
#endif
#define RADIX_MOD(x)        ((x) & ((1 << RADIX_LOG2) - 1))
#define RADIX_DIV(x)        ((x) >> RADIX_LOG2)
#define RADIX_MASK  ((((crypt_uword_t)1) << RADIX_LOG2) - 1)
#define BITS_TO_CRYPT_WORDS(bits)       RADIX_DIV((bits) + (RADIX_BITS - 1))
#define BYTES_TO_CRYPT_WORDS(bytes)     BITS_TO_CRYPT_WORDS(bytes * 8)
#define SIZE_IN_CRYPT_WORDS(thing)      BYTES_TO_CRYPT_WORDS(sizeof(thing))
#if RADIX_BITS == 64
#define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_64(x)
    typedef uint64_t    crypt_uword_t;
    typedef int64_t     crypt_word_t;
#   define TO_CRYPT_WORD_64             BIG_ENDIAN_BYTES_TO_UINT64
#   define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
#elif RADIX_BITS == 32
#   define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_32((x))
    typedef uint32_t    crypt_uword_t;
    typedef int32_t     crypt_word_t;
#   define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h)                             \
        BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h),                                 \
        BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
#endif
#define MAX_CRYPT_UWORD (~((crypt_uword_t)0))
#define MAX_CRYPT_WORD  ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
#define MIN_CRYPT_WORD  (~MAX_CRYPT_WORD)
#define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES),                      \
                            MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
#define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
#define MAX_ECC_PARAMETER_BYTES (MAX_ECC_KEY_BYTES * ALG_ECC)
```

These are the basic big number formats. This is convertible to the library- specific format without to much difficulty. For the math performed using these numbers, the value is always positive.

```
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;
48
49  // The Functions to access the properties of a big number.
50  // Get number of allocated words
51  #define BnGetAllocated(x)   (unsigned)((x)->allocated)
```

Get number of words used

```
52  #define BnGetSize(x)        ((x)->size)
```

Get a pointer to the data array

```
53  #define BnGetArray(x)       ((crypt_uword_t *)&((x)->d[0]))
```

Get the nth word of a BIGNUM (zero-based)

```
54  #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

```
55  #define BnEqualZero(bn)   (BnGetSize(bn) == 0)
```

Test to see if a bignum_t is equal to a word type

```
56  #define BnEqualWord(bn, word)                                                \
57              ((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.

```
58  #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*.

```
59  #define BN_INIT(name)                                                        \
60      (bigNum)BnInit((bigNum)&(name),                                          \
61                     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_*

```
62  #define BN_ADDRESS(name) (bigNum)&name##_
63  #define BN_STRUCT_ALLOCATION(bits) (BITS_TO_CRYPT_WORDS(bits) + 1)
```

Create a structure of the correct size.

```
64  #define BN_STRUCT(bits)                                                    \
65      BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))
```

Define a BIGNUM type with a specific allocation

```
66  #define BN_TYPE(name, bits)                                                \
67      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.

```
68  #define BN_INITIALIZED(name, bits, initializer)                            \
69      BN_STRUCT(bits)  name##_;                                              \
70      bigNum           name = BnFrom2B(BN_INIT(name##_),                     \
71                                       (const TPM2B *)initializer)
```

Create a local variable that can hold a number with *bits*

```
72  #define BN_VAR(name, bits)                                                 \
73      BN_STRUCT(bits)  _##name;                                              \
74      bigNum           name = BN_INIT(_##name)
```

Create a type that can hold the largest number defined by the implementation.

```
75  #define BN_MAX(name)   BN_VAR(name, LARGEST_NUMBER_BITS)
76  #define BN_MAX_INITIALIZED(name, initializer)                              \
77      BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
```

A word size value is useful

```
78  #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.

```
79  #define BN_WORD_INITIALIZED(name, initial)                                 \
80      BN_STRUCT(RADIX_BITS)  name##_;                                        \
81      bigNum                 name = BnInitializeWord((bigNum)&name##_,       \
82                                    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

```
83  typedef struct _bn_point_t
84  {
85      bigNum          x;
86      bigNum          y;
87      bigNum          z;
88  } bn_point_t;
89  typedef bn_point_t          *bigPoint;
90  typedef const bn_point_t    *pointConst;
91  typedef struct constant_point_t
92  {
93      bigConst        x;
94      bigConst        y;
95      bigConst        z;
96  } constant_point_t;
97  #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)                                        \
```

```
150             +    ((UINT64)(e) << 24)                                    \
151             +    ((UINT64)(f) << 16)                                    \
152             +    ((UINT64)(g) << 8)                                     \
153             +    ((UINT64)(h))                                          \
154             )
155 #ifndef RADIX_BYTES
156 #   if RADIX_BITS == 32
157 #       define RADIX_BYTES 4
158 #   elif RADIX_BITS == 64
159 #       define RADIX_BYTES 8
160 #   else
161 #       error "RADIX_BITS must either be 32 or 64"
162 #   endif
163 #endif
```

Add implementation dependent definitions for other ECC Values and for linkages.

```
164 #include LIB_INCLUDE(MATH_LIB, Math)
165 #endif // _BN_NUMBERS_H
```

10.1.2 CryptEcc.h

10.1.2.1 Introduction

This file contains structure definitions used for ECC. The structures in this file are only used internally. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
1   #ifndef _CRYPT_ECC_H
2   #define _CRYPT_ECC_H
```

10.1.2.2 Structures

This is used to define the macro that may or may not be in the data set for the curve (CryptEccData.c). If there is a mismatch, the compiler will warn that there is to much/not enough initialization data in the curve. The macro is used because not all versions of the CryptEccData.c need the curve name.

```
3   #ifdef NAMED_CURVES
4   #define CURVE_NAME(a) , a
5   #define CURVE_NAME_DEF const char *name;
6   #else
7   #  define CURVE_NAME(a)
8   #  define CURVE_NAME_DEF
9   #endif
10  typedef struct ECC_CURVE
11  {
12      const TPM_ECC_CURVE          curveId;
13      const UINT16                 keySizeBits;
14      const TPMT_KDF_SCHEME        kdf;
15      const TPMT_ECC_SCHEME        sign;
16      const ECC_CURVE_DATA        *curveData; // the address of the curve data
17      const BYTE                  *OID;
18      CURVE_NAME_DEF
19  } ECC_CURVE;
20  extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT];
21
22  #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.

```
#ifndef _CRYPT_HASH_H
#define _CRYPT_HASH_H
```

10.1.3.2 Hash-related Structures

```
union SMAC_STATES;
```

These definitions add the high-level methods for processing state that may be an SMAC

```
typedef void(* SMAC_DATA_METHOD)(
    union SMAC_STATES       *state,
    UINT32                   size,
    const BYTE              *buffer
    );
typedef UINT16(* SMAC_END_METHOD)(
    union SMAC_STATES       *state,
    UINT32                   size,
    BYTE                    *buffer
    );
typedef struct sequenceMethods {
    SMAC_DATA_METHOD          data;
    SMAC_END_METHOD           end;
} SMAC_METHODS;
#define SMAC_IMPLEMENTED (CC_MAC || CC_MAC_Start)
```

These definitions are here because the SMAC state is in the union of hash states.

```
typedef struct tpmCmacState {
    TPM_ALG_ID              symAlg;
    UINT16                  keySizeBits;
    INT16                   bcount; // current count of bytes accumulated in IV
    TPM2B_IV                iv;     // IV buffer
    TPM2B_SYM_KEY           symKey;
} tpmCmacState_t;
typedef union SMAC_STATES {
#if ALG_CMAC
    tpmCmacState_t          cmac;
#endif
    UINT64                  pad;
} SMAC_STATES;
typedef struct SMAC_STATE {
    SMAC_METHODS            smacMethods;
    SMAC_STATES             state;
} SMAC_STATE;
typedef union
{
#if ALG_SHA1
    tpmHashStateSHA1_t         Sha1;
#endif
#if ALG_SHA256
    tpmHashStateSHA256_t       Sha256;
#endif
```

```
44  #if ALG_SHA384
45      tpmHashStateSHA384_t       Sha384;
46  #endif
47  #if ALG_SHA512
48      tpmHashStateSHA512_t       Sha512;
49  #endif
50  #if ALG_SM3_256
51      tpmHashStateSM3_256_t      Sm3_256;
52  #endif
53
54  // Additions for symmetric block cipher MAC
55  #if SMAC_IMPLEMENTED
56      SMAC_STATE                 smac;
57  #endif
58      // to force structure alignment to be no worse than HASH_ALIGNMENT
59  #if HASH_ALIGNMENT == 4
60      uint32_t             align;
61  #else
62      uint64_t             align;
63  #endif
64  } ANY_HASH_STATE;
65  typedef ANY_HASH_STATE *PANY_HASH_STATE;
66  typedef const ANY_HASH_STATE    *PCANY_HASH_STATE;
67  #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

```
68  #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
69  #define MAX_HASH_STATE_SIZE_ALIGNED                                    \
70                      ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)
```

This is an aligned byte array that will hold any of the hash contexts.

```
71  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

```
72  #ifndef HASH_START_METHOD_DEF
73  #   define HASH_START_METHOD_DEF    void (HASH_START_METHOD)(void)
74  #endif
75  #ifndef HASH_DATA_METHOD_DEF
76  #   define HASH_DATA_METHOD_DEF     void (HASH_DATA_METHOD)(void)
77  #endif
78  #ifndef HASH_END_METHOD_DEF
79  #   define HASH_END_METHOD_DEF      void (HASH_END_METHOD)(void)
80  #endif
81  #ifndef HASH_STATE_COPY_METHOD_DEF
82  #   define HASH_STATE_COPY_METHOD_DEF     void (HASH_STATE_COPY_METHOD)(void)
83  #endif
84  #ifndef  HASH_STATE_EXPORT_METHOD_DEF
85  #   define  HASH_STATE_EXPORT_METHOD_DEF   void (HASH_STATE_EXPORT_METHOD)(void)
86  #endif
87  #ifndef  HASH_STATE_IMPORT_METHOD_DEF
88  #   define  HASH_STATE_IMPORT_METHOD_DEF   void (HASH_STATE_IMPORT_METHOD)(void)
89  #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.

```
typedef HASH_START_METHOD_DEF;
typedef HASH_DATA_METHOD_DEF;
typedef HASH_END_METHOD_DEF;
typedef HASH_STATE_COPY_METHOD_DEF;
typedef HASH_STATE_EXPORT_METHOD_DEF;
typedef HASH_STATE_IMPORT_METHOD_DEF;
typedef struct _HASH_METHODS
{
    HASH_START_METHOD           *start;
    HASH_DATA_METHOD            *data;
    HASH_END_METHOD             *end;
    HASH_STATE_COPY_METHOD      *copy;      // Copy a hash block
    HASH_STATE_EXPORT_METHOD    *copyOut;   // Copy a hash block from a hash
                                            // context
    HASH_STATE_IMPORT_METHOD    *copyIn;    // Copy a hash block to a proper hash
                                            // context
} HASH_METHODS, *PHASH_METHODS;
#if ALG_SHA1
    TPM2B_TYPE(SHA1_DIGEST, SHA1_DIGEST_SIZE);
#endif
#if ALG_SHA256
    TPM2B_TYPE(SHA256_DIGEST, SHA256_DIGEST_SIZE);
#endif
#if ALG_SHA384
    TPM2B_TYPE(SHA384_DIGEST, SHA384_DIGEST_SIZE);
#endif
#if ALG_SHA512
    TPM2B_TYPE(SHA512_DIGEST, SHA512_DIGEST_SIZE);
#endif
#if ALG_SM3_256
    TPM2B_TYPE(SM3_256_DIGEST, SM3_256_DIGEST_SIZE);
#endif
```

When the TPM implements RSA, the hash-dependent OID pointers are part of the HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the HASH_DEF_TEMPLATE.

```
#if ALG_RSA
#define PKCS1_HASH_REF   const BYTE  *PKCS1;
#define PKCS1_OID(NAME)  , OID_PKCS1_##NAME
#else
#define PKCS1_HASH_REF
#define PKCS1_OID(NAME)
#endif
```

When the TPM implements ECC, the hash-dependent OID pointers are part of the HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the HASH_DEF_TEMPLATE.

```
#if ALG_ECDSA
#define ECDSA_HASH_REF    const BYTE  *ECDSA;
#define ECDSA_OID(NAME)  , OID_ECDSA_##NAME
#else
#define ECDSA_HASH_REF
#define ECDSA_OID(NAME)
#endif
typedef const struct HASH_DEF
{
    HASH_METHODS         method;
    uint16_t             blockSize;
    uint16_t             digestSize;
    uint16_t             contextSize;
    uint16_t             hashAlg;
    const BYTE          *OID;
    PKCS1_HASH_REF      // PKCS1 OID
    ECDSA_HASH_REF      // ECDSA OID
```

```
} HASH_DEF, *PHASH_DEF;
```

Macro to fill in the HASH_DEF for an algorithm. For SHA1, the instance would be: HASH_DEF_TEMPLATE(Sha1, SHA1) This handles the difference in capitalization for the various pieces.

```
#define HASH_DEF_TEMPLATE(HASH, Hash)                                          \
    HASH_DEF    Hash##_Def= {                                                  \
                        {(HASH_START_METHOD *)&tpmHashStart_##HASH,            \
                         (HASH_DATA_METHOD *)&tpmHashData_##HASH,              \
                         (HASH_END_METHOD *)&tpmHashEnd_##HASH,                \
                         (HASH_STATE_COPY_METHOD *)&tpmHashStateCopy_##HASH,   \
                         (HASH_STATE_EXPORT_METHOD *)&tpmHashStateExport_##HASH,\
                         (HASH_STATE_IMPORT_METHOD *)&tpmHashStateImport_##HASH,\
                        },                                                     \
                        HASH##_BLOCK_SIZE,      /*block size */               \
                        HASH##_DIGEST_SIZE,     /*data size */                \
                        sizeof(tpmHashState##HASH##_t),                        \
                        TPM_ALG_##HASH, OID_##HASH                             \
                        PKCS1_OID(HASH) ECDSA_OID(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;
```

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.

```
181   typedef struct
182   {
183       BYTE                    buffer[sizeof(HASH_STATE)];
184   } EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
185   typedef const EXPORT_HASH_STATE *PCEXPORT_HASH_STATE;
186   #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      AES_MAX_KEY_SIZE_BITS
#define DRBG_IV_SIZE_BITS       (AES_MAX_BLOCK_SIZE * 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;
92
93  // When instrumenting RSA key sieve
94  #if  RSA_INSTRUMENT
```

```
95   #define PRIME_INDEX(x)  ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
96   #   define INSTRUMENT_SET(a, b) ((a) = (b))
97   #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
98   #   define INSTRUMENT_INC(a)    (a) = (a) + 1
99   extern UINT32  PrimeIndex;
100  extern UINT32  failedAtIteration[10];
101  extern UINT32  PrimeCounts[3];
102  extern UINT32  MillerRabinTrials[3];
103  extern UINT32  totalFieldsSieved[3];
104  extern UINT32  bitsInFieldAfterSieve[3];
105  extern UINT32  emptyFieldsSieved[3];
106  extern UINT32  noPrimeFields[3];
107  extern UINT32  primesChecked[3];
108  extern UINT16  lastSievePrime;
109  #else
110  #   define INSTRUMENT_SET(a, b)
111  #   define INSTRUMENT_ADD(a, b)
112  #   define INSTRUMENT_INC(a)
113  #endif
114  #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
```

These values are used in the *bigNum* representation of various RSA values.

```
BN_TYPE(rsa, MAX_RSA_KEY_BITS);
#define BN_RSA(name)        BN_VAR(name, MAX_RSA_KEY_BITS)
#define BN_RSA_INITIALIZED(name, initializer)                       \
    BN_INITIALIZED(name, MAX_RSA_KEY_BITS, initializer)
#define BN_PRIME(name)      BN_VAR(name, (MAX_RSA_KEY_BITS / 2))
BN_TYPE(prime, (MAX_RSA_KEY_BITS / 2));
#define BN_PRIME_INITIALIZED(name, initializer)                     \
    BN_INITIALIZED(name, MAX_RSA_KEY_BITS / 2, initializer)
#if !CRT_FORMAT_RSA
#   error   This verson only works with CRT formatted data
#endif // !CRT_FORMAT_RSA
typedef struct privateExponent
{
    bigNum              P;
    bigNum              Q;
    bigNum              dP;
    bigNum              dQ;
    bigNum              qInv;
    bn_prime_t          entries[5];
} privateExponent;
#define     NEW_PRIVATE_EXPONENT(X)                                                 \
    privateExponent          _##X;                                                  \
    privateExponent         *X = RsaInitializeExponent(&(_##X))
#endif      // _CRYPT_RSA_H
```

10.1.6 CryptTest.h

This file contains constant definitions used for self-test.

```
#ifndef _CRYPT_TEST_H
#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.

```
#define ALGORITHM_VECTOR_BYTES  ((ALG_LAST_VALUE + 8) / 8)
typedef BYTE    ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
#ifdef  TEST_SELF_TEST
LIB_EXPORT    extern  ALGORITHM_VECTOR    LibToTest;
#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.
typedef struct
{
    UINT32      rng;
    UINT32      hash;
    UINT32      sym;
#if ALG_RSA
    UINT32      rsa;
#endif
#if ALG_ECC
    UINT32      ecc;
#endif
} CRYPTO_SELF_TEST_STATE;

#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
```

```
59    TPM2B_TYPE(SHA512, 64);
60    TPM2B_SHA512    c_SHA512_digest = {{64, {
61      0xe2,0x7b,0x10,0x3d,0x5e,0x48,0x58,0x44,0x67,0xac,0xa3,0x81,0x8c,0x1d,0xc5,0x71,
62      0x66,0x92,0x8a,0x89,0xaa,0xd4,0x35,0x51,0x60,0x37,0x31,0xd7,0xba,0xe7,0x93,0x0b,
63      0x16,0x4d,0xb3,0xc8,0x34,0x98,0x3c,0xd3,0x53,0xde,0x5e,0xe8,0x0c,0xbc,0xaf,0xc9,
64      0x24,0x2c,0xcc,0xed,0xdb,0xde,0xba,0x1f,0x14,0x14,0x5a,0x95,0x80,0xde,0x66,0xbd
65      }}};
66    #endif
```

10.1.8 KdfTestData.h

Hash Test Vectors

```
#define TEST_KDF_KEY_SIZE   20
TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
TPM2B_KDF_TEST_KEY      c_kdfTestKeyIn = {{TEST_KDF_KEY_SIZE, {
    0x27, 0x1F, 0xA0, 0x8B, 0xBD, 0xC5, 0x06, 0x0E, 0xC3, 0xDF,
    0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C }}};

TPM2B_TYPE(KDF_TEST_LABEL, 17);
TPM2B_KDF_TEST_LABEL    c_kdfTestLabel = {{17, {
    0x4B, 0x44, 0x46, 0x53, 0x45, 0x4C, 0x46, 0x54,
    0x45, 0x53, 0x54, 0x4C, 0x41, 0x42, 0x45, 0x4C, 0x00 }}};

TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextU = {{8, {
    0xCE, 0x24, 0x4F, 0x39, 0x5D, 0xCA, 0x73, 0x91 }}};

TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextV = {{8, {
    0xDA, 0x50, 0x40, 0x31, 0xDD, 0xF1, 0x2E, 0x83 }}};

#if ALG_SHA512 == ALG_YES
    TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
        0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
        0x59, 0xf5, 0x85, 0x7c, 0x56, 0xd6, 0x84, 0x81, 0x0f, 0xd3 }}};
    #define KDF_TEST_ALG    TPM_ALG_SHA512

#elif ALG_SHA384 == ALG_YES
    TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
        0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
        0xd9, 0xcf, 0xc7, 0x2b, 0xf4, 0x6f, 0x45, 0xb0, 0x93, 0x12 }}};
    #define KDF_TEST_ALG    TPM_ALG_SHA384

#elif ALG_SHA256 == ALG_YES
    TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
        0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
        0xd7, 0x04, 0xb6, 0x9a, 0x90, 0x2e, 0x9a, 0xde, 0x84, 0xc4 }}};
    #define KDF_TEST_ALG    TPM_ALG_SHA256

#elif ALG_SHA1 == ALG_YES
    TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
        0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
        0xae, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
    #define KDF_TEST_ALG    TPM_ALG_SHA1
#endif
```

10.1.9 RsaTestData.h

RSA Test Vectors

```
#define RSA_TEST_KEY_SIZE   256
typedef struct
{
    UINT16      size;
    BYTE        buffer[RSA_TEST_KEY_SIZE];
} TPM2B_RSA_TEST_KEY;
typedef TPM2B_RSA_TEST_KEY  TPM2B_RSA_TEST_VALUE;
typedef struct
{
    UINT16       size;
    BYTE        buffer[RSA_TEST_KEY_SIZE / 2];
} TPM2B_RSA_TEST_PRIME;
const TPM2B_RSA_TEST_KEY    c_rsaPublicModulus = {256, {
  0x91,0x12,0xf5,0x07,0x9d,0x5f,0x6b,0x1c,0x90,0xf6,0xcc,0x87,0xde,0x3a,0x7a,0x15,
  0xdc,0x54,0x07,0x6c,0x26,0x8f,0x25,0xef,0x7e,0x66,0xc0,0xe3,0x82,0x12,0x2f,0xab,
  0x52,0x82,0x1e,0x85,0xbc,0x53,0xba,0x2b,0x01,0xad,0x01,0xc7,0x8d,0x46,0x4f,0x7d,
  0xdd,0x7e,0xdc,0xb0,0xad,0xf6,0x0c,0xa1,0x62,0x92,0x97,0x8a,0x3e,0x6f,0x7e,0x3e,
  0xf6,0x9a,0xcc,0xf9,0xa9,0x86,0x77,0xb6,0x85,0x43,0x42,0x04,0x13,0x65,0xe2,0xad,
  0x36,0xc9,0xbf,0xc1,0x97,0x84,0x6f,0xee,0x7c,0xda,0x58,0xd2,0xae,0x07,0x00,0xaf,
  0xc5,0x5f,0x4d,0x3a,0x98,0xb0,0xed,0x27,0x7c,0xc2,0xce,0x26,0x5d,0x87,0xe1,0xe3,
  0xa9,0x69,0x88,0x4f,0x8c,0x08,0x31,0x18,0xae,0x93,0x16,0xe3,0x74,0xde,0xd3,0xf6,
  0x16,0xaf,0xa3,0xac,0x37,0x91,0x8d,0x10,0xc6,0x6b,0x64,0x14,0x3a,0xd9,0xfc,0xe4,
  0xa0,0xf2,0xd1,0x01,0x37,0x4f,0x4a,0xeb,0xe5,0xec,0x98,0xc5,0xd9,0x4b,0x30,0xd2,
  0x80,0x2a,0x5a,0x18,0x5a,0x7d,0xd4,0x3d,0xb7,0x62,0x98,0xce,0x6d,0xa2,0x02,0x6e,
  0x45,0xaa,0x95,0x73,0xe0,0xaa,0x75,0x57,0xb1,0x3d,0x1b,0x05,0x75,0x23,0x6b,0x20,
  0x69,0x9e,0x14,0xb0,0x7f,0xac,0xae,0xd2,0xc7,0x48,0x3b,0xe4,0x56,0x11,0x34,0x1e,
  0x05,0x1a,0x30,0x20,0xef,0x68,0x93,0x6b,0x9d,0x7e,0xdd,0xba,0x96,0x50,0xcc,0x1c,
  0x81,0xb4,0x59,0xb9,0x74,0x36,0xd9,0x97,0xdc,0x8f,0x17,0x82,0x72,0xb3,0x59,0xf6,

0x23,0xfa,0x84,0xf7,0x6d,0xf2,0x05,0xff,0xf1,0xb9,0xcc,0xe9,0xa2,0x82,0x01,0xfb}};

const TPM2B_RSA_TEST_PRIME    c_rsaPrivatePrime = {RSA_TEST_KEY_SIZE / 2, {
  0xb7,0xa0,0x90,0xc7,0x92,0x09,0xde,0x71,0x03,0x37,0x4a,0xb5,0x2f,0xda,0x61,0xb8,
  0x09,0x1b,0xba,0x99,0x70,0x45,0xc1,0x0b,0x15,0x12,0x71,0x8a,0xb3,0x2a,0x4d,0x5a,
  0x41,0x9b,0x73,0x89,0x80,0x0a,0x8f,0x18,0x4c,0x8b,0xa2,0x5b,0xda,0xbd,0x43,0xbe,
  0xdc,0x76,0x4d,0x71,0x0f,0xb9,0xfc,0x7a,0x09,0xfe,0x4f,0xac,0x63,0xd9,0x2e,0x50,
  0x3a,0xa1,0x37,0xc6,0xf2,0xa1,0x89,0x12,0xe7,0x72,0x64,0x2b,0xba,0xc1,0x1f,0xca,
  0x9d,0xb7,0xaa,0x3a,0xa9,0xd3,0xa6,0x6f,0x73,0x02,0xbb,0x85,0x5d,0x9a,0xb9,0x5c,
  0x08,0x83,0x22,0x20,0x49,0x91,0x5f,0x4b,0x86,0xbc,0x3f,0x76,0x43,0x08,0x97,0xbf,

0x82,0x55,0x36,0x2d,0x8b,0x6e,0x9e,0xfb,0xc1,0x67,0x6a,0x43,0xa2,0x46,0x81,0x71}};

const BYTE        c_RsaTestValue[RSA_TEST_KEY_SIZE] = {
  0x2a,0x24,0x3a,0xbb,0x50,0x1d,0xd4,0x2a,0xf9,0x18,0x32,0x34,0xa2,0x0f,0xea,0x5c,
  0x91,0x77,0xe9,0xe1,0x09,0x83,0xdc,0x5f,0x71,0x64,0x5b,0xeb,0x57,0x79,0xa0,0x41,
  0xc9,0xe4,0x5a,0x0b,0xf4,0x9f,0xdb,0x84,0x04,0xa6,0x48,0x24,0xf6,0x3f,0x66,0x1f,
  0xa8,0x04,0x5c,0xf0,0x7a,0x6b,0x4a,0x9c,0x7e,0x21,0xb6,0xda,0x6b,0x65,0x9c,0x3a,
  0x68,0x50,0x13,0x1e,0xa4,0xb7,0xca,0xec,0xd3,0xcc,0xb2,0x9b,0x8c,0x87,0xa4,0x6a,
  0xba,0xc2,0x06,0x3f,0x40,0x48,0x7b,0xa8,0xb8,0x2c,0x03,0x14,0x33,0xf3,0x1d,0xe9,
  0xbd,0x6f,0x54,0x66,0xb4,0x69,0x5e,0xbc,0x80,0x7c,0xe9,0x6a,0x43,0x7f,0xb8,0x6a,
  0xa0,0x5f,0x5d,0x7a,0x20,0xfd,0x7a,0x39,0xe1,0xea,0x0e,0x94,0x91,0x28,0x63,0x7a,
  0xac,0xc9,0xa5,0x3a,0x6d,0x31,0x7b,0x7c,0x54,0x56,0x99,0x56,0xbb,0xb7,0xa1,0x2d,
  0xd2,0x5c,0x91,0x5f,0x1c,0xd3,0x06,0x7f,0x34,0x53,0x2f,0x4c,0xd1,0x8b,0xd2,0x9e,
  0xdc,0xc3,0x94,0x0a,0xe1,0x0f,0xa5,0x15,0x46,0x2a,0x8e,0x10,0xc2,0xfe,0xb7,0x5e,
  0x2d,0x0d,0xd1,0x25,0xfc,0xe4,0xf7,0x02,0x19,0xfe,0xb6,0xe4,0x95,0x9c,0x17,0x4a,
  0x9b,0xdb,0xab,0xc7,0x79,0xe3,0x5e,0x40,0xd0,0x56,0x6d,0x25,0x0a,0x72,0x65,0x80,
  0x92,0x9a,0xa8,0x07,0x70,0x32,0x14,0xfb,0xfe,0x08,0xeb,0x13,0xb4,0x07,0x68,0xb4,
  0x58,0x39,0xbe,0x8e,0x78,0x3a,0x59,0x3f,0x9c,0x4c,0xe9,0xa8,0x64,0x68,0xf7,0xb9,
```

```
57
      0x6e,0x20,0xf5,0xcb,0xca,0x47,0xf2,0x17,0xaa,0x8b,0xbc,0x13,0x14,0x84,0xf6,0xab};
58
59    const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
60      0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
61      0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
62      0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
63      0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
64      0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
65      0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
66      0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
67      0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
68      0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
69      0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
70      0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
71      0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
72      0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
73      0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
74      0x7d,0x65,0xd6,0x5d,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
75
      0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
76
77    #define    OAEP_TEST_LABEL     "OAEP Test Value"
78
79    #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
80
81    const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
82      0x32,0x68,0x84,0x0b,0x9c,0xc9,0x25,0x26,0xd9,0xc0,0xd0,0xb1,0xde,0x60,0x55,0xae,
83      0x33,0xe5,0xcf,0x6c,0x85,0xbe,0x0d,0x71,0x11,0xe1,0x45,0x60,0xbb,0x42,0x3d,0xf3,
84      0xb1,0x18,0x84,0x7b,0xc6,0x5d,0xce,0x1d,0x5f,0x9a,0x97,0xcf,0xb1,0x97,0x9a,0x85,
85      0x7c,0xa7,0xa1,0x63,0x23,0xb6,0x74,0x0f,0x1a,0xee,0x29,0x51,0xeb,0x50,0x8f,0x3c,
86      0x8e,0x4e,0x31,0x38,0xdc,0x11,0xfc,0x9a,0x4e,0xaf,0x93,0xc9,0x7f,0x6e,0x35,0xf3,
87      0xc9,0xe4,0x89,0x14,0x53,0xe2,0xc2,0x1a,0xf7,0x6b,0x9b,0xf0,0x7a,0xa4,0x69,0x52,
88      0xe0,0x24,0x8f,0xea,0x31,0xa7,0x5c,0x43,0xb0,0x65,0xc9,0xfe,0xba,0xfe,0x80,0x9e,
89      0xa5,0xc0,0xf5,0x8d,0xce,0x41,0xf9,0x83,0x0d,0x8e,0x0f,0xef,0x3d,0x1f,0x6a,0xcc,
90      0x8a,0x3d,0x3b,0xdf,0x22,0x38,0xd7,0x34,0x58,0x7b,0x55,0xc9,0xf6,0xbc,0x7c,0x4c,
91      0x3f,0xd7,0xde,0x4e,0x30,0xa9,0x69,0xf3,0x5f,0x56,0x8f,0xc2,0xe7,0x75,0x79,0xb8,
92      0xa5,0xc8,0x0d,0xc0,0xcd,0xb6,0xc9,0x63,0xad,0x7c,0xe4,0x8f,0x39,0x60,0x4d,0x7d,
93      0xdb,0x34,0x49,0x2a,0x47,0xde,0xc0,0x42,0x4a,0x19,0x94,0x2e,0x50,0x21,0x03,0x47,
94      0xff,0x73,0xb3,0xb7,0x89,0xcc,0x7b,0x2c,0xeb,0x03,0xa7,0x9a,0x06,0xfd,0xed,0x19,
95      0xbb,0x82,0xa0,0x13,0xe9,0xfa,0xac,0x06,0x5f,0xc5,0xa9,0x2b,0xda,0x88,0x23,0xa2,
96      0x5d,0xc2,0x7f,0xda,0xc8,0x5a,0x94,0x31,0xc1,0x21,0xd7,0x1e,0x6b,0xd7,0x89,0xb1,
97
      0x93,0x80,0xab,0xd1,0x37,0xf2,0x6f,0x50,0xcd,0x2a,0xea,0xb1,0xc4,0xcd,0xcb,0xb5}};
98
99    const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
100     0x29,0xa4,0x2f,0xbb,0x8a,0x14,0x05,0x1e,0x3c,0x72,0x76,0x77,0x38,0xe7,0x73,0xe3,
101     0x6e,0x24,0x4b,0x38,0xd2,0x1a,0xcf,0x23,0x58,0x78,0x36,0x82,0x23,0x6e,0x6b,0xef,
102     0x2c,0x3d,0xf2,0xe8,0xd6,0xc6,0x87,0x8e,0x78,0x9b,0x27,0x39,0xc0,0xd6,0xef,0x4d,
103     0x0b,0xfc,0x51,0x27,0x18,0xf3,0x51,0x5e,0x4d,0x96,0x3a,0xe2,0x15,0xe2,0x7e,0x42,
104     0xf4,0x16,0xd5,0xc6,0x52,0x5d,0x17,0x44,0x76,0x09,0x7a,0xcf,0xe3,0x30,0xe3,0x84,
105     0xf6,0x6f,0x3a,0x33,0xfb,0x32,0x0d,0x1d,0xe7,0x7c,0x80,0x82,0x4f,0xed,0xda,0x87,
106     0x11,0x9c,0xc3,0x7e,0x85,0xbd,0x18,0x58,0x08,0x2b,0x23,0x37,0xe7,0x9d,0xd0,0xd1,
107     0x79,0xe2,0x05,0xbd,0xf5,0x4f,0x0e,0x0f,0xdb,0x4a,0x74,0xeb,0x09,0x01,0xb3,0xca,
108     0xbd,0xa6,0x7b,0x09,0xb1,0x13,0x77,0x30,0x4d,0x87,0x41,0x06,0x57,0x2e,0x5f,0x36,
109     0x6e,0xfc,0x35,0x69,0xfe,0x0a,0x24,0x6c,0x98,0x8c,0xda,0x97,0xf4,0xfb,0xc7,0x83,
110     0x2d,0x3e,0x7d,0xc0,0x5c,0x34,0xfd,0x11,0x2a,0x12,0xa7,0xae,0x4a,0xde,0xc8,0x4e,
111     0xcf,0xf4,0x85,0x63,0x77,0xc6,0x33,0x34,0xe0,0x27,0xe4,0x9e,0x91,0x0b,0x4b,0x85,
112     0xf0,0xb0,0x79,0xaa,0x7c,0xc6,0xff,0x3b,0xbc,0x04,0x73,0xb8,0x95,0xd7,0x31,0x54,
113     0x3b,0x56,0xec,0x52,0x15,0xd7,0x3e,0x62,0xf5,0x82,0x99,0x3e,0x2a,0xc0,0x4b,0x2e,
114     0x06,0x57,0x6d,0x3f,0x3e,0x77,0x1f,0x2b,0x2d,0xc5,0xb9,0x3b,0x68,0x56,0x73,0x70,
115
      0x32,0x6b,0x6b,0x65,0x25,0x76,0x45,0x6c,0x45,0xf1,0x6c,0x59,0xfc,0x94,0xa7,0x15}};
```

```
116
117   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
118     0x01,0xfe,0xd5,0x83,0x0b,0x15,0xba,0x90,0x2c,0xdf,0xf7,0x26,0xb7,0x8f,0xb1,0xd7,
119     0x0b,0xfd,0x83,0xf9,0x95,0xd5,0xd7,0xb5,0xc5,0xc5,0x4a,0xde,0xd5,0xe6,0x20,0x78,
120     0xca,0x73,0x77,0x3d,0x61,0x36,0x48,0xae,0x3e,0x8f,0xee,0x43,0x29,0x96,0xdf,0x3f,
121     0x1c,0x97,0x5a,0xbe,0xe5,0xa2,0x7e,0x5b,0xd0,0xc0,0x29,0x39,0x83,0x81,0x77,0x24,
122     0x43,0xdb,0x3c,0x64,0x4d,0xf0,0x23,0xe4,0xae,0x0f,0x78,0x31,0x8c,0xda,0x0c,0xec,
123     0xf1,0xdf,0x09,0xf2,0x14,0x6a,0x4d,0xaf,0x36,0x81,0x6e,0xbd,0xbe,0x36,0x79,0x88,
124     0x98,0xb6,0x6f,0x5a,0xad,0xcf,0x7c,0xee,0xe0,0xdd,0x00,0xbe,0x59,0x97,0x88,0x00,
125     0x34,0xc0,0x8b,0x48,0x42,0x05,0x04,0x5a,0xb7,0x85,0x38,0xa0,0x35,0xd7,0x3b,0x51,
126     0xb8,0x7b,0x81,0x83,0xee,0xff,0x76,0x6f,0x50,0x39,0x4d,0xab,0x89,0x63,0x07,0x6d,
127     0xf5,0xe5,0x01,0x10,0x56,0xfe,0x93,0x06,0x8f,0xd3,0xc9,0x41,0xab,0xc9,0xdf,0x6e,
128     0x59,0xa8,0xc3,0x1d,0xbf,0x96,0x4a,0x59,0x80,0x3c,0x90,0x3a,0x59,0x56,0x4c,0x6d,
129     0x44,0x6d,0xeb,0xdc,0x73,0xcd,0xc1,0xec,0xb8,0x41,0xbf,0x89,0x8c,0x03,0x69,0x4c,
130     0xaf,0x3f,0xc1,0xc5,0xc7,0xe7,0x7d,0xa7,0x83,0x39,0x70,0xa2,0x6b,0x83,0xbc,0xbe,
131     0xf5,0xbf,0x1c,0xee,0x6e,0xa3,0x22,0x1e,0x25,0x2f,0x16,0x68,0x69,0x5a,0x1d,0xfa,
132     0x2c,0x3a,0x0f,0x67,0xe1,0x77,0x12,0xe8,0x3d,0xba,0xaa,0xef,0x96,0x9c,0x1f,0x64,
133
      0x32,0xf4,0xa7,0xb3,0x3f,0x7d,0x61,0xbb,0x9a,0x27,0xad,0xfb,0x2f,0x33,0xc4,0x70}};
134
135   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
136     0x67,0x4e,0xdd,0xc2,0xd2,0x6d,0xe0,0x03,0xc4,0xc2,0x41,0xd3,0xd4,0x61,0x30,0xd0,
137     0xe1,0x68,0x31,0x4a,0xda,0xd9,0xc2,0x5d,0xaa,0xa2,0x7b,0xfb,0x44,0x02,0xf5,0xd6,
138     0xd8,0x2e,0xcd,0x13,0x36,0xc9,0x4b,0xdb,0x1a,0x4b,0x66,0x1b,0x4f,0x9c,0xb7,0x17,
139     0xac,0x53,0x37,0x4f,0x21,0xbd,0x0c,0x66,0xac,0x06,0x65,0x52,0x9f,0x04,0xf6,0xa5,
140     0x22,0x5b,0xf7,0xe6,0x0d,0x3c,0x9f,0x41,0x19,0x09,0x88,0x7c,0x41,0x4c,0x2f,0x9c,
141     0x8b,0x3c,0xdd,0x7c,0x28,0x78,0x24,0xd2,0x09,0xa6,0x5b,0xf7,0x3c,0x88,0x7e,0x73,
142     0x5a,0x2d,0x36,0x02,0x4f,0x65,0xb0,0xcb,0xc8,0xdc,0xac,0xa2,0xda,0x8b,0x84,0x91,
143     0x71,0xe4,0x30,0x8b,0xb6,0x12,0xf2,0xf0,0xd0,0xa0,0x38,0xcf,0x75,0xb7,0x20,0xcb,
144     0x35,0x51,0x52,0x6b,0xc4,0xf4,0x21,0x95,0xc2,0xf7,0x9a,0x13,0xc1,0x1a,0x7b,0x8f,
145     0x77,0xda,0x19,0x48,0xbb,0x6d,0x14,0x5d,0xba,0x65,0xb4,0x9e,0x43,0x42,0x58,0x98,
146     0x0b,0x91,0x46,0xd8,0x4c,0xf3,0x4c,0xaf,0x2e,0x02,0xa6,0xb2,0x49,0x12,0x62,0x43,
147     0x4e,0xa8,0xac,0xbf,0xfd,0xfa,0x37,0x24,0xea,0x69,0x1c,0xf5,0xae,0xfa,0x08,0x82,
148     0x30,0xc3,0xc0,0xf8,0x9a,0x89,0x33,0xe1,0x40,0x6d,0x18,0x5c,0x7b,0x90,0x48,0xbf,
149     0x37,0xdb,0xea,0xfb,0x0e,0xd4,0x2e,0x11,0xfa,0xa9,0x86,0xff,0x00,0x0b,0x7b,0xca,
150     0x09,0x64,0x6a,0x8f,0x0c,0x0e,0x09,0x14,0x36,0x4a,0x74,0x31,0x18,0x5b,0x18,0xeb,
151
      0xea,0x83,0xc3,0x66,0x68,0xa6,0x7d,0x43,0x06,0x0f,0x99,0x60,0xce,0x65,0x08,0xf6}};
152
153   #endif // SHA1
154
155   #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
156
157   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
158     0x33,0x20,0x6e,0x21,0xc3,0xf6,0xcd,0xf8,0xd7,0x5d,0x9f,0xe9,0x05,0x14,0x8c,0x7c,
159     0xbb,0x69,0x24,0x9e,0x52,0x8f,0xaf,0x84,0x73,0x21,0x2c,0x85,0xa5,0x30,0x4d,0xb6,
160     0xb8,0xfa,0x15,0x9b,0xc7,0x8f,0xc9,0x7a,0x72,0x4b,0x85,0xa4,0x1c,0xc5,0xd8,0xe4,
161     0x92,0xb3,0xec,0xd9,0xa8,0xca,0x5e,0x74,0x73,0x89,0x7f,0xb4,0xac,0x7e,0x68,0x12,
162     0xb2,0x53,0x27,0x4b,0xbf,0xd0,0x71,0x69,0x46,0x9f,0xef,0xf4,0x70,0x60,0xf8,0xd7,
163     0xae,0xc7,0x5a,0x27,0x38,0x25,0x2d,0x25,0xab,0x96,0x56,0x66,0x3a,0x23,0x40,0xa8,
164     0xdb,0xbc,0x86,0xe8,0xf3,0xd2,0x58,0x0b,0x44,0xfc,0x94,0x1e,0xb7,0x5d,0xb4,0x57,
165     0xb5,0xf3,0x56,0xee,0x9b,0xcf,0x97,0x91,0x29,0x36,0xe3,0x06,0x13,0xa2,0xea,0xd6,
166     0xd6,0x0b,0x86,0x0b,0x1a,0x27,0xe6,0x22,0xc4,0x7b,0xff,0xde,0x0f,0xbf,0x79,0xc8,
167     0x1b,0xed,0xf1,0x27,0x62,0xb5,0x8b,0xf9,0xd9,0x76,0x90,0xf6,0xcc,0x83,0x0f,0xce,
168     0xce,0x2e,0x63,0x7a,0x9b,0xf4,0x48,0x5b,0xd7,0x81,0x2c,0x3a,0xdb,0x59,0x0d,0x4d,
169     0x9e,0x46,0xe9,0x9e,0x92,0x22,0x27,0x1c,0xb0,0x67,0x8a,0xe6,0x8a,0x16,0x8a,0xdf,
170     0x95,0x76,0x24,0x82,0xad,0xf1,0xbc,0x97,0xbf,0xd3,0x5e,0x6e,0x14,0x0c,0x5b,0x25,
171     0xfe,0x58,0xfa,0x64,0xe5,0x14,0x46,0xb7,0x58,0xc6,0x3f,0x7f,0x42,0xd2,0x8e,0x45,
172     0x13,0x41,0x85,0x12,0x2e,0x96,0x19,0xd0,0x5e,0x7d,0x34,0x06,0x32,0x2b,0xc8,0xd9,
173
      0x0d,0x6c,0x06,0x36,0xa0,0xff,0x47,0x57,0x2c,0x25,0xbc,0x8a,0xa5,0xe2,0xc7,0xe3}};
174
175   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
```

```
176     0x39,0xfc,0x10,0x5d,0xf4,0x45,0x3d,0x94,0x53,0x06,0x89,0x24,0xe7,0xe8,0xfd,0x03,
177     0xac,0xfd,0xbd,0xb2,0x28,0xd3,0x4a,0x52,0xc5,0xd4,0xdb,0x17,0xd4,0x24,0x05,0xc4,
178     0xeb,0x6a,0xce,0x1d,0xbb,0x37,0xcb,0x09,0xd8,0x6c,0x83,0x19,0x93,0xd4,0xe2,0x88,
179     0x88,0x9b,0xaf,0x92,0x16,0xc4,0x15,0xbd,0x49,0x13,0x22,0xb7,0x84,0xcf,0x23,0xf2,
180     0x6f,0x0c,0x3e,0x8f,0xde,0x04,0x09,0x31,0x2d,0x99,0xdf,0xe6,0x74,0x70,0x30,0xde,
181     0x8c,0xad,0x32,0x86,0xe2,0x7c,0x12,0x90,0x21,0xf3,0x86,0xb7,0xe2,0x64,0xca,0x98,
182     0xcc,0x64,0x4b,0xef,0x57,0x4f,0x5a,0x16,0x6e,0xd7,0x2f,0x5b,0xf6,0x07,0xad,0x33,
183     0xb4,0x8f,0x3b,0x3a,0x8b,0xd9,0x06,0x2b,0xed,0x3c,0x3c,0x76,0xf6,0x21,0x31,0xe3,
184     0xfb,0x2c,0x45,0x61,0x42,0xba,0xe0,0xc3,0x72,0x63,0xd0,0x6b,0x8f,0x36,0x26,0xfb,
185     0x9e,0x89,0x0e,0x44,0x9a,0xc1,0x84,0x5e,0x84,0x8d,0xb6,0xea,0xf1,0x0d,0x66,0xc7,
186     0xdb,0x44,0xbd,0x19,0x7c,0x05,0xbe,0xc4,0xab,0x88,0x32,0xbe,0xc7,0x63,0x31,0xe6,
187     0x38,0xd4,0xe5,0xb8,0x4b,0xf5,0x0e,0x55,0x9a,0x3a,0xe6,0x0a,0xec,0xee,0xe2,0xa8,
188     0x88,0x04,0xf2,0xb8,0xaa,0x5a,0xd8,0x97,0x5d,0xa0,0xa8,0x42,0xfb,0xd9,0xde,0x80,
189     0xae,0x4c,0xb3,0xa1,0x90,0x47,0x57,0x03,0x10,0x78,0xa6,0x8f,0x11,0xba,0x4b,0xce,
190     0x2d,0x56,0xa4,0xe1,0xbd,0xf8,0xa0,0xa4,0xd5,0x48,0x3c,0x63,0x20,0x00,0x38,0xa0,
191
    0xd1,0xe6,0x12,0xe9,0x1d,0xd8,0x49,0xe3,0xd5,0x24,0xb5,0xc5,0x3a,0x1f,0xb0,0xd4}};
192
193 const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
194     0x74,0x89,0x29,0x3e,0x1b,0xac,0xc6,0x85,0xca,0xf0,0x63,0x43,0x30,0x7d,0x1c,0x9b,
195     0x2f,0xbd,0x4d,0x69,0x39,0x5e,0x85,0xe2,0xef,0x86,0x0a,0xc6,0x6b,0xa6,0x08,0x19,
196     0x6c,0x56,0x38,0x24,0x55,0x92,0x84,0x9b,0x1b,0x8b,0x04,0xcf,0x24,0x14,0x24,0x13,
197     0x0e,0x8b,0x82,0x6f,0x96,0xc8,0x9a,0x68,0xfc,0x4c,0x02,0xf0,0xdc,0xcd,0x36,0x25,
198     0x31,0xd5,0x82,0xcf,0xc9,0x69,0x72,0xf6,0x1d,0xab,0x68,0x20,0x2e,0x2d,0x19,0x49,
199     0xf0,0x2e,0xad,0xd2,0xda,0xaf,0xff,0xb6,0x92,0x83,0x5b,0x8a,0x06,0x2d,0x0c,0x32,
200     0x11,0x32,0x3b,0x77,0x17,0xf6,0x50,0xfb,0xf8,0x57,0xc9,0xc7,0x9b,0x9e,0xc6,0xd1,
201     0xa9,0x55,0xf0,0x22,0x35,0xda,0xca,0x3c,0x8e,0xc6,0x9a,0xd8,0x25,0xc8,0x5e,0x93,
202     0x0d,0xaa,0xa7,0x06,0xaf,0x11,0x29,0x99,0xe7,0x7c,0xee,0x49,0x82,0x30,0xba,0x2c,
203     0xe2,0x40,0x8f,0x0a,0xa6,0x7b,0x24,0x75,0xc5,0xcd,0x03,0x12,0xf4,0xb2,0x4b,0x3a,
204     0xd1,0x91,0x3c,0x20,0x0e,0x58,0x2b,0x31,0xf8,0x8b,0xee,0xbc,0x1f,0x95,0x35,0x58,
205     0x6a,0x73,0xee,0x99,0xb0,0x01,0x42,0x4f,0x66,0xc0,0x66,0xbb,0x35,0x86,0xeb,0xd9,
206     0x7b,0x55,0x77,0x2d,0x54,0x78,0x19,0x49,0xe8,0xcc,0xfd,0xb1,0xcb,0x49,0xc9,0xea,
207     0x20,0xab,0xed,0xb5,0xed,0xfe,0xb2,0xb5,0xa8,0xcf,0x05,0x06,0xd5,0x7d,0x2b,0xbb,
208     0x0b,0x65,0x6b,0x2b,0x6d,0x55,0x95,0x85,0x44,0x8b,0x12,0x05,0xf3,0x4b,0xd4,0x8e,
209
    0x3d,0x68,0x2d,0x29,0x9c,0x05,0x79,0xd6,0xfc,0x72,0x90,0x6a,0xab,0x46,0x38,0x81}};
210
211 const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
212     0x8a,0xb1,0x0a,0xb5,0xe4,0x02,0xf7,0xdd,0x45,0x2a,0xcc,0x2b,0x6b,0x8c,0x0e,0x9a,
213     0x92,0x4f,0x9b,0xc5,0xe4,0x8b,0x82,0xb9,0xb0,0xd9,0x87,0x8c,0xcb,0xf0,0xb0,0x59,
214     0xa5,0x92,0x21,0xa0,0xa7,0x61,0x5c,0xed,0xa8,0x6e,0x22,0x29,0x46,0xc7,0x86,0x37,
215     0x4b,0x1b,0x1e,0x94,0x93,0xc8,0x4c,0x17,0x7a,0xae,0x59,0x91,0xf8,0x83,0x84,0xc4,
216     0x8c,0x38,0xc2,0x35,0x0e,0x7e,0x50,0x67,0x76,0xe7,0xd3,0xec,0x6f,0x0d,0xa0,0x5c,
217     0x2f,0x0a,0x80,0x28,0xd3,0xc5,0x7d,0x2d,0x1a,0x0b,0x96,0xd6,0xe5,0x98,0x05,0x8c,
218     0x4d,0xa0,0x1f,0x8c,0xb6,0xfb,0xb1,0xcf,0xe9,0xcb,0x38,0x27,0x60,0x64,0x17,0xca,
219     0xf4,0x8b,0x61,0xb7,0x1d,0xb6,0x20,0x9d,0x40,0x2a,0x1c,0xfd,0x55,0x40,0x4b,0x95,
220     0x39,0x52,0x18,0x3b,0xab,0x44,0xe8,0x83,0x4b,0x7c,0x47,0xfb,0xed,0x06,0x9c,0xcd,
221     0x4f,0xba,0x81,0xd6,0xb7,0x31,0xcf,0x5c,0x23,0xf8,0x25,0xab,0x95,0x77,0x0a,0x8f,
222     0x46,0xef,0xfb,0x59,0xb8,0x04,0xd7,0x1e,0xf5,0xaf,0x6a,0x1a,0x26,0x9b,0xae,0xf4,
223     0xf5,0x7f,0x84,0x6f,0x3c,0xed,0xf8,0x24,0x0b,0x43,0xd1,0xba,0x74,0x89,0x4e,0x39,
224     0xfe,0xab,0xa5,0x16,0xa5,0x28,0xee,0x96,0x84,0x3e,0x16,0x6d,0x5f,0x4e,0x0b,0x7d,
225     0x94,0x16,0x1b,0x8c,0xf9,0xaa,0x9b,0xc0,0x49,0x02,0x4c,0x3e,0x62,0xff,0xfe,0xa2,
226     0x20,0x33,0x5e,0xa6,0xdd,0xda,0x15,0x2d,0xb7,0xcd,0xda,0xff,0xb1,0x0b,0x45,0x7b,
227
    0xd3,0xa0,0x42,0x29,0xab,0xa9,0x73,0xe9,0xa4,0xd9,0x8d,0xac,0xa1,0x88,0x2c,0x2d}};
228
229 #endif // SHA256
230
231 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
232
233 const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
234     0x0f,0x3c,0x42,0x4d,0x8c,0x91,0x96,0x05,0x3c,0xfd,0x59,0x3b,0x7f,0x29,0xbc,0x03,
235     0x67,0xc1,0xff,0x74,0xe7,0x09,0xf4,0x13,0x45,0xbe,0x13,0x1d,0xc9,0x86,0x94,0xfe,
```

```
236     0xed,0xa6,0xe8,0x3a,0xcb,0x89,0x4d,0xec,0x86,0x63,0x4c,0xdb,0xf1,0x95,0xee,0xc1,
237     0x46,0xc5,0x3b,0xd8,0xf8,0xa2,0x41,0x6a,0x60,0x8b,0x9e,0x5e,0x7f,0x20,0x16,0xe3,
238     0x69,0xb6,0x2d,0x92,0xfc,0x60,0xa2,0x74,0x88,0xd5,0xc7,0xa6,0xd1,0xff,0xe3,0x45,
239     0x02,0x51,0x39,0xd9,0xf3,0x56,0x0b,0x91,0x80,0xe0,0x6c,0xa8,0xc3,0x78,0xef,0x34,
240     0x22,0x8c,0xf5,0xfb,0x47,0x98,0x5d,0x57,0x8e,0x3a,0xb9,0xff,0x92,0x04,0xc7,0xc2,
241     0x6e,0xfa,0x14,0xc1,0xb9,0x68,0x15,0x5c,0x12,0xe8,0xa8,0xbe,0xea,0xe8,0x8d,0x9b,
242     0x48,0x28,0x35,0xdb,0x4b,0x52,0xc1,0x2d,0x85,0x47,0x83,0xd0,0xe9,0xae,0x90,0x6e,
243     0x65,0xd4,0x34,0x7f,0x81,0xce,0x69,0xf0,0x96,0x62,0xf7,0xec,0x41,0xd5,0xc2,0xe3,
244     0x4b,0xba,0x9c,0x8a,0x02,0xce,0xf0,0x5d,0x14,0xf7,0x09,0x42,0x8e,0x4a,0x27,0xfe,
245     0x3e,0x66,0x42,0x99,0x03,0xe1,0x69,0xbd,0xdb,0x7f,0x9b,0x70,0xeb,0x4e,0x9c,0xac,
246     0x45,0x67,0x91,0x9f,0x75,0x10,0xc6,0xfc,0x14,0xe1,0x28,0xc1,0x0e,0xe0,0x7e,0xc0,
247     0x5c,0x1d,0xee,0xe8,0xff,0x45,0x79,0x51,0x86,0x08,0xe6,0x39,0xac,0xb5,0xfd,0xb8,
248     0xf1,0xdd,0x2e,0xf4,0xb2,0x1a,0x69,0x0d,0xd9,0x98,0x8e,0xdb,0x85,0x61,0x70,0x20,
249
        0x82,0x91,0x26,0x87,0x80,0xc4,0x6a,0xd8,0x3b,0x91,0x4d,0xd3,0x33,0x84,0xad,0xb7}};
250
251   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
252     0x44,0xd5,0x9f,0xbc,0x48,0x03,0x3d,0x9f,0x22,0x91,0x2a,0xab,0x3c,0x31,0x71,0xab,
253     0x86,0x3f,0x0f,0x6f,0x59,0x5b,0x93,0x27,0xbc,0xbc,0xcd,0x29,0x38,0x43,0x2a,0x3b,
254     0x3b,0xd2,0xb3,0x45,0x40,0xba,0x15,0xb4,0x45,0xe3,0x56,0xab,0xff,0xb3,0x20,0x26,
255     0x39,0xcc,0x48,0xc5,0x5d,0x41,0x0d,0x2f,0x57,0x7f,0x9d,0x16,0x2e,0x26,0x57,0xc7,
256     0x6b,0xf3,0x36,0x54,0xbd,0xb6,0x1d,0x46,0x4e,0x13,0x50,0xd7,0x61,0x9d,0x8d,0x7b,
257     0xeb,0x21,0x9f,0x79,0xf3,0xfd,0xe0,0x1b,0xa8,0xed,0x6d,0x29,0x33,0x0d,0x65,0x94,
258     0x24,0x1e,0x62,0x88,0x6b,0x2b,0x4e,0x39,0xf5,0x80,0x39,0xca,0x76,0x95,0xbc,0x7c,
259     0x27,0x1d,0xdd,0x3a,0x11,0xf1,0x3e,0x54,0x03,0xb7,0x43,0x91,0x99,0x33,0xfe,0x9d,
260     0x14,0x2c,0x87,0x9a,0x95,0x18,0x1f,0x02,0x04,0x6a,0xe2,0xb7,0x81,0x14,0x13,0x45,
261     0x16,0xfb,0xe4,0xb7,0x8f,0xab,0x2b,0xd7,0x60,0x34,0x8a,0x55,0xbc,0x01,0x8c,0x49,
262     0x02,0x29,0xf1,0x9c,0x94,0x98,0x44,0xd0,0x94,0xcb,0xd4,0x85,0x4c,0x3b,0x77,0x72,
263     0x99,0xd5,0x4b,0xc6,0x3b,0xe4,0xd2,0xc8,0xe9,0x6a,0x23,0x18,0x3b,0x3b,0x5e,0x32,
264     0xec,0x70,0x84,0x5d,0xbb,0x6a,0x8f,0x0c,0x5f,0x55,0xa5,0x30,0x34,0x48,0xbb,0xc2,
265     0xdf,0x12,0xb9,0x81,0xad,0x36,0x3f,0xf0,0x24,0x16,0x48,0x04,0x4a,0x7f,0xfd,0x9f,
266     0x4c,0xea,0xfe,0x1d,0x83,0xd0,0x81,0xad,0x25,0x6c,0x5f,0x45,0x36,0x91,0xf0,0xd5,
267
        0x8b,0x53,0x0a,0xdf,0xec,0x9f,0x04,0x58,0xc4,0x35,0xa0,0x78,0x1f,0x68,0xe0,0x22}};
268
269   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
270     0x3f,0x3a,0x82,0x6d,0x42,0xe3,0x8b,0x4f,0x45,0x9c,0xda,0x6c,0xbe,0xbe,0xcd,0x00,
271     0x98,0xfb,0xbe,0x59,0x30,0xc6,0x3c,0xaa,0xb3,0x06,0x27,0xb5,0xda,0xfa,0xb2,0xc3,
272     0x43,0xb7,0xbd,0xe9,0xd3,0x23,0xed,0x80,0xce,0x74,0xb3,0xb8,0x77,0x8d,0xe6,0x8d,
273     0x3c,0xe5,0xf5,0xd7,0x80,0xcf,0x38,0x55,0x76,0xd7,0x87,0xa8,0xd6,0x3a,0xcf,0xfd,
274     0xd8,0x91,0x65,0xab,0x43,0x66,0x50,0xb7,0x9a,0x13,0x6b,0x45,0x80,0x76,0x86,0x22,
275     0x27,0x72,0xf7,0xbb,0x65,0x22,0x5c,0x55,0x60,0xd8,0x84,0x9f,0xf2,0x61,0x52,0xac,
276     0xf2,0x4f,0x5b,0x7b,0x21,0xe1,0xf5,0x4b,0x8f,0x01,0xf2,0x4b,0xcf,0xd3,0xfb,0x74,
277     0x5e,0x6e,0x96,0xb4,0xa8,0x0f,0x01,0x9b,0x26,0x54,0x0a,0x70,0x55,0x26,0xb7,0x0b,
278     0xe8,0x01,0x68,0x66,0x0d,0x6f,0xb5,0xfc,0x66,0xbd,0x9e,0x44,0xed,0x6a,0x1e,0x3c,
279     0x3b,0x61,0x5d,0xe8,0xdb,0x99,0x5b,0x67,0xbf,0x94,0xfb,0xe6,0x8c,0x4b,0x07,0xcb,
280     0x43,0x3a,0x0d,0xb1,0x1b,0x10,0x66,0x81,0xe2,0x0d,0xe7,0xd1,0xca,0x85,0xa7,0x50,
281     0x82,0x2d,0xbf,0xed,0xcf,0x43,0x6d,0xdb,0x2c,0x7b,0x73,0x20,0xfe,0x73,0x3f,0x19,
282     0xc6,0xdb,0x69,0xb8,0xc3,0xd3,0xf4,0xe5,0x64,0xf8,0x36,0x8e,0xd5,0xd8,0x09,0x2a,
283     0x5f,0x26,0x70,0xa1,0xd9,0x5b,0x14,0xf8,0x22,0xe9,0x9d,0x22,0x51,0xf4,0x52,0xc1,
284     0x6f,0x53,0xf5,0xca,0x0d,0xda,0x39,0x8c,0x29,0x42,0xe8,0x58,0x89,0xbb,0xd1,0x2e,
285
        0xc5,0xdb,0x86,0x8d,0xaf,0xec,0x58,0x36,0x8d,0x8d,0x57,0x23,0xd5,0xdd,0xb9,0x24}};
286
287   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
288     0x39,0x10,0x58,0x7d,0x6d,0xa8,0xd5,0x90,0x07,0xd6,0x2b,0x13,0xe9,0xd8,0x93,0x7e,
289     0xf3,0x5d,0x71,0xe0,0xf0,0x33,0x3a,0x4a,0x22,0xf3,0xe6,0x95,0xd3,0x8e,0x8c,0x41,
290     0xe7,0xb3,0x13,0xde,0x4a,0x45,0xd3,0xd1,0xfb,0xb1,0x3f,0x9b,0x39,0xa5,0x50,0x58,
291     0xef,0xb6,0x3a,0x43,0xdd,0x54,0xab,0xda,0x9d,0x32,0x49,0xe4,0x57,0x96,0xe5,0x1b,
292     0x1d,0x8f,0x33,0x8e,0x07,0x67,0x56,0x14,0xc1,0x18,0x78,0xa2,0x52,0xe6,0x2e,0x07,
293     0x81,0xbe,0xd8,0xca,0x76,0x63,0x68,0xc5,0x47,0xa2,0x92,0x5e,0x4c,0xfd,0x14,0xc7,
294     0x46,0x14,0xbe,0xc7,0x85,0xef,0xe6,0xb8,0x46,0xcb,0x3a,0x67,0x66,0x89,0xc6,0xee,
295     0x9d,0x64,0xf5,0x0d,0x09,0x80,0x9a,0x6f,0x0e,0xeb,0xe4,0xb9,0xe9,0xab,0x90,0x4f,
```

```
296       0xe7,0x5a,0xc8,0xca,0xf6,0x16,0x0a,0x82,0xbd,0xb7,0x76,0x59,0x08,0x2d,0xd9,0x40,
297       0x5d,0xaa,0xa5,0xef,0xfb,0xe3,0x81,0x2c,0x2c,0x5c,0xa8,0x16,0xbd,0x63,0x20,0xc2,
298       0x4d,0x3b,0x51,0xaa,0x62,0x1f,0x06,0xe5,0xbb,0x78,0x44,0x04,0x0c,0x5c,0xe1,0x1b,
299       0x6b,0x9d,0x21,0x10,0xaf,0x48,0x48,0x98,0x97,0x77,0xc2,0x73,0xb4,0x98,0x64,0xcc,
300       0x94,0x2c,0x29,0x28,0x45,0x36,0xd1,0xc5,0xd0,0x2f,0x97,0x27,0x92,0x65,0x22,0xbb,
301       0x63,0x79,0xea,0xf5,0xff,0x77,0x0f,0x4b,0x56,0x8a,0x9f,0xad,0x1a,0x97,0x67,0x39,
302       0x69,0xb8,0x4c,0x6c,0xc2,0x56,0xc5,0x7a,0xa8,0x14,0x5a,0x24,0x7a,0xa4,0x6e,0x55,
303
          0xb2,0x86,0x1d,0xf4,0x62,0x5a,0x2d,0x87,0x6d,0xde,0x99,0x78,0x2d,0xef,0xd7,0xdc}};
304
305   #endif // SHA384
306
307   #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
308
309   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
310     0x48,0x45,0xa7,0x70,0xb2,0x41,0xb7,0x48,0x5e,0x79,0x8c,0xdf,0x1c,0xc6,0x7e,0xbb,
311     0x11,0x80,0x82,0x52,0xbf,0x40,0x3d,0x90,0x03,0x6e,0x20,0x3a,0xb9,0x65,0xc8,0x51,
312     0x4c,0xbd,0x9c,0xa9,0x43,0x89,0xd0,0x57,0x0c,0xa3,0x69,0x22,0x7e,0x82,0x2a,0x1c,
313     0x1d,0x5a,0x80,0x84,0x81,0xbb,0x5e,0x5e,0xd0,0xc1,0x66,0x9a,0xac,0x00,0xba,0x14,
314     0xa2,0xe9,0xd0,0x3a,0x89,0x5a,0x63,0xe2,0xec,0x92,0x05,0xf4,0x47,0x66,0x12,0x7f,
315     0xdb,0xa7,0x3c,0x5b,0x67,0xe1,0x55,0xca,0x0a,0x27,0xbf,0x39,0x89,0x11,0x05,0xba,
316     0x9b,0x5a,0x9b,0x65,0x44,0xad,0x78,0xcf,0x8f,0x94,0xf6,0x9a,0xb4,0x52,0x39,0x0e,
317     0x00,0xba,0xbc,0xe0,0xbd,0x6f,0x81,0x2d,0x76,0x42,0x66,0x70,0x07,0x77,0xbf,0x09,
318     0x88,0x2a,0x0c,0xb1,0x56,0x3e,0xee,0xfd,0xdc,0xb6,0x3c,0x0d,0xc5,0xa4,0x0d,0x10,
319     0x32,0x80,0x3e,0x1e,0xfe,0x36,0x8f,0xb5,0x42,0xc1,0x21,0x7b,0xdf,0xdf,0x4a,0xd2,
320     0x68,0x0c,0x01,0x9f,0x4a,0xfd,0xd4,0xec,0xf7,0x49,0x06,0xab,0xed,0xc6,0xd5,0x1b,
321     0x63,0x76,0x38,0xc8,0x6c,0xc7,0x4f,0xcb,0x29,0x8a,0x0e,0x6f,0x33,0xaf,0x69,0x31,
322     0x8e,0xa7,0xdd,0x9a,0x36,0xde,0x9b,0xf1,0x0b,0xfb,0x20,0xa0,0x6d,0x33,0x31,0xc9,
323     0x9e,0xb4,0x2e,0xc5,0x40,0x0e,0x60,0x71,0x36,0x75,0x05,0xf9,0x37,0xe0,0xca,0x8e,
324     0x8f,0x56,0xe0,0xea,0x9b,0xeb,0x17,0xf3,0xca,0x40,0xc3,0x48,0x01,0xba,0xdc,0xc6,
325
          0x4b,0x2b,0x5b,0x7b,0x5c,0x81,0xa6,0xbb,0xc7,0x43,0xc0,0xbe,0xc0,0x30,0x7b,0x55}};
326
327   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
328     0x74,0x83,0xfa,0x52,0x65,0x50,0x68,0xd0,0x82,0x05,0x72,0x70,0x78,0x1c,0xac,0x10,
329     0x23,0xc5,0x07,0xf8,0x93,0xd2,0xeb,0x65,0x87,0xbb,0x47,0xc2,0xfb,0x30,0x9e,0x61,
330     0x4c,0xac,0x04,0x57,0x5a,0x7c,0xeb,0x29,0x08,0x84,0x86,0x89,0x1e,0x8f,0x07,0x32,
331     0xa3,0x8b,0x70,0xe7,0xa2,0x9f,0x9c,0x42,0x71,0x3d,0x23,0x59,0x82,0x5e,0x8a,0xde,
332     0xd6,0xfb,0xd8,0xc5,0x8b,0xc0,0xdb,0x10,0x38,0x87,0xd3,0xbf,0x04,0xb0,0x66,0xb9,
333     0x85,0x81,0x54,0x4c,0x69,0xdc,0xba,0x78,0xf3,0x4a,0xdb,0x25,0xa2,0xf2,0x34,0x55,
334     0xdd,0xaa,0xa5,0xc4,0xed,0x55,0x06,0x0e,0x2a,0x30,0x77,0xab,0x82,0x79,0xf0,0xcd,
335     0x9d,0x6f,0x09,0xa0,0xc8,0x82,0xc9,0xe0,0x61,0xda,0x40,0xcd,0x17,0x59,0xc0,0xef,
336     0x95,0x6d,0xa3,0x6d,0x1c,0x2b,0xee,0x24,0xef,0xd8,0x4a,0x55,0x6c,0xd6,0x26,0x42,
337     0x32,0x17,0xfd,0x6a,0xb3,0x4f,0xde,0x07,0x2f,0x10,0xd4,0xac,0x14,0xea,0x89,0x68,
338     0xcc,0xd3,0x07,0xb7,0xcf,0xba,0x39,0x20,0x63,0x20,0x7b,0x44,0x8b,0x48,0x60,0x5d,
339     0x3a,0x2a,0x0a,0xe9,0x68,0xab,0x15,0x46,0x27,0x64,0xb5,0x82,0x06,0x29,0xe7,0x25,
340     0xca,0x46,0x48,0x6e,0x2a,0x34,0x57,0x4b,0x81,0x75,0xae,0xb6,0xfd,0x6f,0x51,0x5f,
341     0x04,0x59,0xc7,0x15,0x1f,0xe0,0x68,0xf7,0x36,0x2d,0xdf,0xc8,0x9d,0x05,0x27,0x2d,
342     0x3f,0x2b,0x59,0x5d,0xcb,0xf3,0xc4,0x92,0x6e,0x00,0xa8,0x8d,0xd0,0x69,0xe5,0x59,
343
          0xda,0xba,0x4f,0x38,0xf5,0xa0,0x8b,0xf1,0x73,0xe9,0x0d,0xee,0x64,0xe5,0xa2,0xd8}};
344
345   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
346     0x1b,0xca,0x8b,0x18,0x15,0x3b,0x95,0x5b,0x0a,0x89,0x10,0x03,0x7f,0x7c,0xa0,0xc9,
347     0x66,0x57,0x86,0x6a,0xc9,0xeb,0x82,0x71,0xf3,0x8d,0x6f,0xa9,0xa4,0x2d,0xd0,0x22,
348     0xdf,0xe9,0xc6,0x71,0x5b,0xf4,0x27,0x38,0x5b,0x2c,0x8a,0x54,0xcc,0x85,0x11,0x69,
349     0x6d,0x6f,0x42,0xe7,0x22,0xcb,0xd6,0xad,0x1a,0xc5,0xab,0x6a,0xa5,0xfc,0xa5,0x70,
350     0x72,0x4a,0x62,0x25,0xd0,0xa2,0x16,0x61,0xab,0xac,0x31,0xa0,0x46,0x24,0x4f,0xdd,
351     0x9a,0x36,0x55,0xb6,0x00,0x9e,0x23,0x50,0x0d,0x53,0x01,0xb3,0x46,0x56,0xb2,0x1d,
352     0x33,0x5b,0xca,0x41,0x7f,0x65,0x7e,0x00,0x5c,0x12,0xff,0x0a,0x70,0x5d,0x8c,0x69,
353     0x4a,0x02,0xee,0x72,0x30,0xa7,0x5c,0xa4,0xbb,0xbe,0x03,0x0c,0xe4,0x5f,0x33,0xb6,
354     0x78,0x91,0x9d,0xd8,0xec,0x34,0x03,0x2e,0x63,0x32,0xc7,0x2a,0x36,0x50,0xd5,0x8b,
355     0x0e,0x7f,0x54,0x4e,0xf4,0x29,0x11,0x1b,0xcd,0x0f,0x37,0xa5,0xbc,0x61,0x83,0x50,
```

```
356     0xfa,0x18,0x75,0xd9,0xfe,0xa7,0xe8,0x9b,0xc1,0x4f,0x96,0x37,0x81,0x71,0xdf,0x71,
357     0x8b,0x89,0x81,0xf4,0x95,0xb5,0x29,0x66,0x41,0x0c,0x73,0xd7,0x0b,0x21,0xb4,0xfb,
358     0xf9,0x63,0x2f,0xe9,0x7b,0x38,0xaa,0x20,0xc3,0x96,0xcc,0xb7,0xb2,0x24,0xa1,0xe0,
359     0x59,0x9c,0x10,0x9e,0x5a,0xf7,0xe3,0x02,0xe6,0x23,0xe2,0x44,0x21,0x3f,0x6e,0x5e,
360     0x79,0xb2,0x93,0x7d,0xce,0xed,0xe2,0xe1,0xab,0x98,0x07,0xa7,0xbd,0xbc,0xd8,0xf7,
361
        0x06,0xeb,0xc5,0xa6,0x37,0x18,0x11,0x88,0xf7,0x63,0x39,0xb9,0x57,0x29,0xdc,0x03}};
362
363     const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
364     0x05,0x55,0x00,0x62,0x01,0xc6,0x04,0x31,0x55,0x73,0x3f,0x2a,0xf9,0xd4,0x0f,0xc1,
365     0x2b,0xeb,0xd8,0xc8,0xdb,0xb2,0xab,0x6c,0x26,0xde,0x2d,0x89,0xc2,0x2d,0x36,0x62,
366     0xc8,0x22,0x5d,0x58,0x03,0xb1,0x46,0x14,0xa5,0xd4,0xbc,0x25,0x6b,0x7f,0x8f,0x14,
367     0x7e,0x03,0x2f,0x3d,0xb8,0x39,0xa5,0x79,0x13,0x7e,0x22,0x2a,0xb9,0x3e,0x8f,0xaa,
368     0x01,0x7c,0x03,0x12,0x21,0x6c,0x2a,0xb4,0x39,0x98,0x6d,0xff,0x08,0x6c,0x59,0x2d,
369     0xdc,0xc6,0xf1,0x77,0x62,0x10,0xa6,0xcc,0xe2,0x71,0x8e,0x97,0x00,0x87,0x5b,0x0e,
370     0x20,0x00,0x3f,0x18,0x63,0x83,0xf0,0xe4,0x0a,0x64,0x8c,0xe9,0x8c,0x91,0xe7,0x89,
371     0x04,0x64,0x2c,0x8b,0x41,0xc8,0xac,0xf6,0x5a,0x75,0xe6,0xa5,0x76,0x43,0xcb,0xa5,
372     0x33,0x8b,0x07,0xc9,0x73,0x0f,0x45,0xa4,0xc3,0xac,0xc1,0xc3,0xe6,0xe7,0x21,0x66,
373     0x1c,0xba,0xbf,0xea,0x3e,0x39,0xfa,0xb2,0xe2,0x8f,0xfe,0x9c,0xb4,0x85,0x89,0x33,
374     0x2a,0x0c,0xc8,0x5d,0x58,0xe1,0x89,0x12,0xe9,0x4d,0x42,0xb3,0x1f,0x99,0x0c,0x3e,
375     0xd8,0xb2,0xeb,0xf5,0x88,0xfb,0xe1,0x4b,0x8e,0xdc,0xd3,0xa8,0xda,0xbe,0x04,0x45,
376     0xbf,0x56,0xc6,0x54,0x70,0x00,0xb8,0x66,0x46,0x3a,0xa3,0x1e,0xb6,0xeb,0x1a,0xa0,
377     0x0b,0xd3,0x9a,0x9a,0x52,0xda,0x60,0x69,0xb7,0xef,0x93,0x47,0x38,0xab,0x1a,0xa0,
378     0x22,0x6e,0x76,0x06,0xb6,0x74,0xaf,0x74,0x8f,0x51,0xc0,0x89,0x5a,0x4b,0xbe,0x6a,
379
        0x91,0x18,0x25,0x7d,0xa6,0x77,0xe6,0xfd,0xc2,0x62,0x36,0x07,0xc6,0xef,0x79,0xc9}};
380
381     #endif // SHA512
```

10.1.10 SelfTest.h

10.1.10.1 Introduction

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

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

10.1.10.2 Defines

Was typing this a lot

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

Use the definition of key sizes to set algorithm values for key size.

```
4     #define AES_ENTRIES (AES_128 + AES_192 + AES_256)
5     #define SM4_ENTRIES (SM4_128)
6     #define CAMELLIA_ENTRIES (CAMELLIA_128 + CAMELLIA_192 + CAMELLIA_256)
7     #define TDES_ENTRIES (TDES_128 + TDES_192)
8     #define NUM_SYMS    (AES_ENTRIES + SM4_ENTRIES + CAMELLIA_ENTRIES + TDES_ENTRIES)
9     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

```
10    #define TPM_SYM_MODE_FIRST      ALG_CTR_VALUE
11    #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
typedef UINT16  SYM_MODES;
#elif NUM_SYM_MODES <= 32
typedef UINT32  SYM_MODES;
#else
#error "Too many symmetric modes"
#endif
typedef struct SYMMETRIC_TEST_VECTOR {
    const TPM_ALG_ID     alg;                   // the algorithm
    const UINT16         keyBits;               // bits in the key
    const BYTE          *key;                   // The test key
    const UINT32         ivSize;                // block size of the algorithm
    const UINT32         dataInOutSize;         // size  to encrypt/decrypt
    const BYTE          *dataIn;                // data to encrypt
    const BYTE          *dataOut[NUM_SYM_MODES];// data to decrypt
} SYMMETRIC_TEST_VECTOR;
#if ALG_SHA512
#       define  DEFAULT_TEST_HASH               ALG_SHA512_VALUE
#       define  DEFAULT_TEST_DIGEST_SIZE        SHA512_DIGEST_SIZE
#       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA512_BLOCK_SIZE
#elif ALG_SHA384
#       define  DEFAULT_TEST_HASH               ALG_SHA384_VALUE
#       define  DEFAULT_TEST_DIGEST_SIZE        SHA384_DIGEST_SIZE
#       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA384_BLOCK_SIZE
#elif ALG_SHA256
#       define  DEFAULT_TEST_HASH               ALG_SHA256_VALUE
#       define  DEFAULT_TEST_DIGEST_SIZE        SHA256_DIGEST_SIZE
#       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA256_BLOCK_SIZE
#elif ALG_SHA1
#       define  DEFAULT_TEST_HASH               ALG_SHA1_VALUE
#       define  DEFAULT_TEST_DIGEST_SIZE        SHA1_DIGEST_SIZE
#       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA1_BLOCK_SIZE
#endif
#endif      // _SELF_TEST_H_
```

10.1.11 SupportLibraryFunctionPrototypes_fp.h

10.1.11.1 Introduction

This file contains the function prototypes for the functions that need to be present in the selected math 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
```

10.1.11.2 SupportLibInit()

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

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

10.1.11.3 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.

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

10.1.11.4 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);
```

10.1.11.5 BnMult()

Multiplies two numbers and returns the result

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

10.1.11.6 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);
```

10.1.11.7 BnMod()

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

10.1.11.8 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);
```


10.1.11.9 BnModExp()

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

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

10.1.11.10 BnModInverse()

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

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

10.1.11.11 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.

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

10.1.11.12 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.

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

10.1.11.13 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.

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

10.1.11.14 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.

```
31  LIB_EXPORT bigCurve
32  BnCurveInitialize(bigCurve E, TPM_ECC_CURVE curveId);
```

10.1.11.14.1 BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```
33  LIB_EXPORT void
34  BnCurveFree(bigCurve E);
35  #endif
```

10.1.12 SymmetricTestData.h

This is a vector for testing either encrypt or decrypt. The premise for decrypt is that the IV for decryption is the same as the IV for encryption. However, the *ivOut* value may be different for encryption and decryption. We will encrypt at least two blocks. This means that the chaining value will be used for each of the schemes (if any) and that implicitly checks that the chaining value is handled properly.

```
#if AES_128
const BYTE  key_AES128 [] = {
                0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
                0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};

const BYTE  dataIn_AES128 [] = {
                0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
                0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};

const BYTE  dataOut_AES128_ECB [] = {
                0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
                0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
                0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
                0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};

const BYTE  dataOut_AES128_CBC [] = {
                0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
                0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
                0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
                0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};

const BYTE  dataOut_AES128_CFB [] = {
                0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
                0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
                0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
                0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};

const BYTE  dataOut_AES128_OFB [] = {
                0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
                0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
                0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
                0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};

const BYTE  dataOut_AES128_CTR [] = {
                0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
                0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
                0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
                0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
#endif

#if AES_192

const BYTE  key_AES192 [] = {
                0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
                0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
                0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};

const BYTE  dataIn_AES192 [] = {
                0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
                0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};

```

```
const BYTE  dataOut_AES192_ECB [] = {
                0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
                0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
                0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
                0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};

const BYTE  dataOut_AES192_CBC [] = {
                0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
                0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
                0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
                0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};

const BYTE  dataOut_AES192_CFB [] = {
                0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
                0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
                0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
                0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};

const BYTE  dataOut_AES192_OFB [] = {
                0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
                0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
                0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
                0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};

const BYTE  dataOut_AES192_CTR [] = {
                0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
                0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
                0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
                0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};
#endif

#if AES_256

const BYTE  key_AES256 [] = {
                0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
                0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
                0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};

const BYTE  dataIn_AES256 [] = {
                0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
                0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};

const BYTE  dataOut_AES256_ECB [] = {
                0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
                0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
                0x59, 0x1c, 0xcb, 0x10, 0xd4, 0x10, 0xed, 0x26,
                0xdc, 0x5b, 0xa7, 0x4a, 0x31, 0x36, 0x28, 0x70};

const BYTE  dataOut_AES256_CBC [] = {
                0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba,
                0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
                0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
                0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};

const BYTE  dataOut_AES256_CFB [] = {
                0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
                0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
                0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
                0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};

```

```
119   const BYTE  dataOut_AES256_OFB [] = {
120                   0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
121                   0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
122                   0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
123                   0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};
124
125   const BYTE  dataOut_AES256_CTR [] = {
126                   0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
127                   0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
128                   0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
129                   0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5};
130   #endif
```

10.1.13 SymmetricTest.h

10.1.13.1 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 may only be included in AlgorithmTests.c"
3   #endif
4   #ifndef     _SYMMETRIC_TEST_H
5   #define     _SYMMETRIC_TEST_H
6   #include    "SymmetricTestData.h"
```

10.1.13.2 Symmetric Test Structures

```
7   const SYMMETRIC_TEST_VECTOR   c_symTestValues[NUM_SYMS + 1] = {
8   #if ALG_AES && AES_128
9       {ALG_AES_VALUE, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128,
10      {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC,
11       dataOut_AES128_CFB, dataOut_AES128_ECB}},
12  #endif
13  #if ALG_AES && AES_192
14      {ALG_AES_VALUE, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192,
15      {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC,
16       dataOut_AES192_CFB, dataOut_AES192_ECB}},
17  #endif
18  #if ALG_AES && AES_256
19      {ALG_AES_VALUE, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256,
20      {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC,
21      dataOut_AES256_CFB, dataOut_AES256_ECB}},
22  #endif
23  // There are no SM4 test values yet so...
24  #if ALG_SM4 && SM4_128 && 0
25      {ALG_SM4_VALUE, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128,
26      {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC,
27       dataOut_SM4128_CFB, dataOut_AES128_ECB}},
28  #endif
29      {0}
30  };
31  #endif  // _SYMMETRIC_TEST_H
```

10.1.14 EccTestData.h

This file contains the parameter data for ECC testing.

```
#ifdef  SELF_TEST_DATA
TPM2B_TYPE(EC_TEST, 32);
const  TPM_ECC_CURVE        c_testCurve = 00003;

// The "static" key

const  TPM2B_EC_TEST    c_ecTestKey_ds = {{32, {
 0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x2d,0xa1,0xb4,0x39,0x7a,
0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}}}
;

const  TPM2B_EC_TEST    c_ecTestKey_QsX = {{32, {
 0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}}}
;

const  TPM2B_EC_TEST    c_ecTestKey_QsY = {{32, {
 0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}}}
;

// The "ephemeral" key

const  TPM2B_EC_TEST    c_ecTestKey_de = {{32, {
 0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
0xc1,0xf8,0xdf,0xdd,0xb8,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xee,0xac}}}
;

const  TPM2B_EC_TEST    c_ecTestKey_QeX = {{32, {
 0xa5,0x1e,0x80,0xd1,0x76,0x3e,0x8b,0x96,0xce,0xcc,0x21,0x82,0xc9,0xa2,0xa2,0xed,
0x47,0x21,0x89,0x53,0x44,0xe9,0xc7,0x92,0xe7,0x31,0x48,0x38,0xe6,0xea,0x93,0x47}}}
;

const  TPM2B_EC_TEST    c_ecTestKey_QeY = {{32, {
 0x30,0xe6,0x4f,0x97,0x03,0xa1,0xcb,0x3b,0x32,0x2a,0x70,0x39,0x94,0xeb,0x4e,0xea,
0x55,0x88,0x81,0x3f,0xb5,0x00,0xb8,0x54,0x25,0xab,0xd4,0xda,0xfd,0x53,0x7a,0x18}}}
;

// ECDH test results
const  TPM2B_EC_TEST    c_ecTestEcdh_X = {{32, {
 0x64,0x02,0x68,0x92,0x78,0xdb,0x33,0x52,0xed,0x3b,0xfa,0x3b,0x74,0xa3,0x3d,0x2c,
0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}}}
;

const  TPM2B_EC_TEST    c_ecTestEcdh_Y = {{32, {
 0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0x89,0xe3,0xe5,
0x10,0x4a,0xbc,0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe6,0xff,0x85,0x11,0x40}}}
;

TPM2B_TYPE(TEST_VALUE, 64);
```

```
43  const TPM2B_TEST_VALUE        c_ecTestValue = {{64, {
44   0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
45   0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
46   0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
47
    0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}}
    ;
48
49  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
50
51  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
52   0x57,0xf3,0x36,0xb7,0xec,0xc2,0xdd,0x76,0x0e,0xe2,0x81,0x21,0x49,0xc5,0x66,0x11,
53
    0x4b,0x8a,0x4f,0x17,0x62,0x82,0xcc,0x06,0xf6,0x64,0x78,0xef,0x6b,0x7c,0xf2,0x6c}}}
    ;
54  const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
55   0x1b,0xed,0x23,0x72,0x8f,0x17,0x5f,0x47,0x2e,0xa7,0x97,0x2c,0x51,0x57,0x20,0x70,
56
    0x6f,0x89,0x74,0x8a,0xa8,0xf4,0x26,0xf4,0x96,0xa1,0xb8,0x3e,0xe5,0x35,0xc5,0x94}}}
    ;
57
58  const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32,{
59   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1b,0x08,0x9f,0xde,
60
    0xef,0x62,0xe3,0xf1,0x14,0xcb,0x54,0x28,0x13,0x76,0xfc,0x6d,0x69,0x22,0xb5,0x3e}}}
    ;
61  const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
62   0xd9,0xd3,0x20,0xfb,0x4d,0x16,0xf2,0xe6,0xe2,0x45,0x07,0x45,0x1c,0x92,0x92,0x92,
63
    0xa9,0x6b,0x48,0xf8,0xd1,0x98,0x29,0x4d,0xd3,0x8f,0x56,0xf2,0xbb,0x2e,0x22,0x3b}}}
    ;
64
65  #endif // SHA1
66
67  #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
68
69  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
70   0x04,0x7d,0x54,0xeb,0x04,0x6f,0x56,0xec,0xa2,0x6c,0x38,0x8c,0xeb,0x43,0x0b,0x71,
71
    0xf8,0xf2,0xf4,0xa5,0xe0,0x1d,0x3c,0xa2,0x39,0x31,0xe4,0xe7,0x36,0x3b,0xb5,0x5f}}}
    ;
72  const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
73   0x8f,0xd0,0x12,0xd9,0x24,0x75,0xf6,0xc4,0x3b,0xb5,0x46,0x75,0x3a,0x41,0x8d,0x80,
74
    0x23,0x99,0x38,0xd7,0xe2,0x40,0xca,0x9a,0x19,0x2a,0xfc,0x54,0x75,0xd3,0x4a,0x6e}}}
    ;
75
76  const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
77   0xf7,0xb9,0x15,0x4c,0x34,0xf6,0x41,0x19,0xa3,0xd2,0xf1,0xbd,0xf4,0x13,0x6a,0x4f,
78
    0x63,0xb8,0x4d,0xb5,0xc8,0xcd,0xde,0x85,0x95,0xa5,0x39,0x0a,0x14,0x49,0x3d,0x2f}}}
    ;
79  const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
80   0xfe,0xbe,0x17,0xaa,0x31,0x22,0x9f,0xd0,0xd2,0xf5,0x25,0x04,0x92,0xb0,0xaa,0x4e,
81
    0xcc,0x1c,0xb6,0x79,0xd6,0x42,0xb3,0x4e,0x3f,0xbb,0xfe,0x5f,0xd0,0xd0,0x8b,0xc3}}}
    ;
82
83  #endif // SHA256
84
85  #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
86
87  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
```

```
88      0xf5,0x74,0x6d,0xd6,0xc6,0x56,0x86,0xbb,0xba,0x1c,0xba,0x75,0x65,0xee,0x64,0x31,
89
        0xce,0x04,0xe3,0x9f,0x24,0x3f,0xbd,0xfe,0x04,0xcd,0xab,0x7e,0xfe,0xad,0xcb,0x82}}}
        ;
90      const TPM2B_EC_TEST     c_TestEcDsa_s = {{32, {
91      0xc2,0x4f,0x32,0xa1,0x06,0xc0,0x85,0x4f,0xc6,0xd8,0x31,0x66,0x91,0x9f,0x79,0xcd,
92
        0x5b,0xe5,0x7b,0x94,0xa1,0x91,0x38,0xac,0xd4,0x20,0xa2,0x10,0xf0,0xd5,0x9d,0xbf}}}
        ;
93
94      const TPM2B_EC_TEST     c_TestEcSchnorr_r = {{32, {
95      0x1e,0xb8,0xe1,0xbf,0xa1,0x9e,0x39,0x1e,0x58,0xa2,0xe6,0x59,0xd0,0x1a,0x6a,0x03,
96
        0x6a,0x1f,0x1c,0x4f,0x36,0x19,0xc1,0xec,0x30,0xa4,0x85,0x1b,0xe9,0x74,0x35,0x66}}}
        ;
97      const TPM2B_EC_TEST     c_TestEcSchnorr_s = {{32,{
98      0xb9,0xe6,0xe3,0x7e,0xcb,0xb9,0xea,0xf1,0xcc,0xf4,0x48,0x44,0x4a,0xda,0xc8,0xd7,
99
        0x87,0xb4,0xba,0x40,0xfe,0x5b,0x68,0x11,0x14,0xcf,0xa0,0x0e,0x85,0x46,0x99,0x01}}}
        ;
100
101     #endif // SHA384
102
103     #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
104
105     const TPM2B_EC_TEST     c_TestEcDsa_r = {{32, {
106     0xc9,0x71,0xa6,0xb4,0xaf,0x46,0x26,0x8c,0x27,0x00,0x06,0x3b,0x00,0x0f,0xa3,0x17,
107
        0x72,0x48,0x40,0x49,0x4d,0x51,0x4f,0xa4,0xcb,0x7e,0x86,0xe9,0xe7,0xb4,0x79,0xb2}}}
        ;
108     const TPM2B_EC_TEST     c_TestEcDsa_s = {{32,{
109     0x87,0xbc,0xc0,0xed,0x74,0x60,0x9e,0xfa,0x4e,0xe8,0x16,0xf3,0xf9,0x6b,0x26,0x07,
110
        0x3c,0x74,0x31,0x7e,0xf0,0x62,0x46,0xdc,0xd6,0x45,0x22,0x47,0x3e,0x0c,0xa0,0x02}}}
        ;
111
112     const TPM2B_EC_TEST     c_TestEcSchnorr_r = {{32,{
113     0xcc,0x07,0xad,0x65,0x91,0xdd,0xa0,0x10,0x23,0xae,0x53,0xec,0xdf,0xf1,0x50,0x90,
114
        0x16,0x96,0xf4,0x45,0x09,0x73,0x9c,0x84,0xb5,0x5c,0x5f,0x08,0x51,0xcb,0x60,0x01}}}
        ;
115     const TPM2B_EC_TEST     c_TestEcSchnorr_s = {{32,{
116     0x55,0x20,0x21,0x54,0xe2,0x49,0x07,0x47,0x71,0xf4,0x99,0x15,0x54,0xf3,0xab,0x14,
117
        0xdb,0x8e,0xda,0x79,0xb6,0x02,0x0e,0xe3,0x5e,0x6f,0x2c,0xb6,0x05,0xbd,0x14,0x10}}}
        ;
118
119     #endif // SHA512
120
121     #endif // SELF_TEST_DATA
```

10.1.15 CryptSym.h

10.1.15.1 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.

10.1.15.2 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
```

10.1.16 OIDs.h

```
#ifndef _OIDS_H_
#define _OIDS_H_
```

All the OIDs in this file are defined as DER-encoded values with a leading tag 0x06 (ASN1_OBJECT_IDENTIFIER), followed by a single length byte. This allows the OID size to be determined by looking at octet[1] of the OID (total size is OID[1] + 2). These macros allow OIDs to be defined (or not) depending on whether the associated hash algorithm is implemented.

NOTE: When one of these macros is used, the NAME needs '_" on each side. The exception is when the macro is used for the hash OID when only a single _ is used.

```
#ifndef ALG_SHA1
#   define ALG_SHA1 NO
#endif
#if ALG_SHA1
#define SHA1_OID(NAME)     MAKE_OID(NAME##SHA1)
#else
#define SHA1_OID(NAME)
#endif
#ifndef ALG_SHA256
#   define ALG_SHA256 NO
#endif
#if ALG_SHA256
#define SHA256_OID(NAME)   MAKE_OID(NAME##SHA256)
#else
#define SHA256_OID(NAME)
#endif
#ifndef ALG_SHA384
#   define ALG_SHA384 NO
#endif
#if ALG_SHA384
#define SHA384_OID(NAME)   MAKE_OID(NAME##SHA384)
#else
#define SHA#84_OID(NAME)
#endif
#ifndef ALG_SHA512
#   define ALG_SHA512 NO
#endif
#if ALG_SHA512
#define SHA512_OID(NAME)   MAKE_OID(NAME##SHA512)
#else
#define SHA512_OID(NAME)
#endif
#ifndef ALG_SM3_256
#   define ALG_SM3_256 NO
#endif
#if ALG_SM3_256
#define SM3_256_OID(NAME) MAKE_OID(NAME##SM3_256)
#else
#define SM3_256_OID(NAME)
#endif
#ifndef ALG_SHA3_256
#   define ALG_SHA3_256 NO
#endif
#if ALG_SHA3_256
#define SHA3_256_OID(NAME) MAKE_OID(NAME##SHA3_256)
#else
#define SHA3_256_OID(NAME)
#endif
#ifndef ALG_SHA3_384
#   define ALG_SHA3_384 NO
#endif
```

```
#if ALG_SHA3_384
#define SHA3_384_OID(NAME) MAKE_OID(NAME##SHA3_384)
#else
#define SHA3_384_OID(NAME)
#endif
#ifndef ALG_SHA3_512
#   define ALG_SHA3_512 NO
#endif
#if ALG_SHA3_512
#define SSHA3_512_OID(NAME) MAKE_OID(NAME##SHA3_512)
#else
#define SHA3_512_OID(NAME)
#endif
```

These are encoded to take one additional byte of algorithm selector

```
#define NIST_HASH       0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 2
#define NIST_SIG        0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 3
```

These hash OIDs used in a lot of places.

```
#define OID_SHA1_VALUE              0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A
SHA1_OID(_);         // Expands to
                     //      MAKE_OID(_SHA1)
                     // which expands to:
                     //      extern BYTE     OID_SHA1[]
                     // or
                     //      const BYTE      OID_SHA1[] = {OID_SHA1_VALUE}
                     // which is:
                     //      const BYTE      OID_SHA1[] = {0x06, 0x05, 0x2B, 0x0E,
                     //                                    0x03, 0x02, 0x1A}
#define OID_SHA256_VALUE            NIST_HASH, 1
SHA256_OID(_);
#define OID_SHA384_VALUE            NIST_HASH, 2
SHA384_OID(_);
#define OID_SHA512_VALUE            NIST_HASH, 3
SHA512_OID(_);
#define OID_SM3_256_VALUE           0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
                                    0x83, 0x11
SM3_256_OID(_);          // (1.2.156.10197.1.401)
#define OID_SHA3_256_VALUE          NIST_HASH, 8
SHA3_256_OID(_);
#define OID_SHA3_384_VALUE          NIST_HASH, 9
SHA3_384_OID(_);
#define OID_SHA3_512_VALUE          NIST_HASH, 10
SHA3_512_OID(_);
```

These are used for RSA-PSS

```
#if ALG_RSA
#define OID_MGF1_VALUE              0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
                                    0x01, 0x01, 0x08
MAKE_OID(_MGF1);
#define OID_RSAPSS_VALUE            0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
                                    0x01, 0x01, 0x0A
MAKE_OID(_RSAPSS);
```

This is the OID to designate the public part of an RSA key.

```
#define OID_PKCS1_PUB_VALUE         0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
                                    0x01, 0x01, 0x01
MAKE_OID(_PKCS1_PUB);
```

These are used for RSA PKCS1 signature Algorithms

```
104  #define OID_PKCS1_SHA1_VALUE        0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,        \
105                                      0x0D, 0x01, 0x01, 0x05
106  SHA1_OID(_PKCS1_);      // (1.2.840.113549.1.1.5)
107  #define OID_PKCS1_SHA256_VALUE      0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,        \
108                                      0x0D, 0x01, 0x01, 0x0B
109  SHA256_OID(_PKCS1_);    // (1.2.840.113549.1.1.11)
110  #define OID_PKCS1_SHA384_VALUE      0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,        \
111                                      0x0D, 0x01, 0x01, 0x0C
112  SHA384_OID(_PKCS1_);    // (1.2.840.113549.1.1.12)
113  #define OID_PKCS1_SHA512_VALUE      0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,        \
114                                      0x0D, 0x01, 0x01, 0x0D
115  SHA512_OID(_PKCS1_);    //(1.2.840.113549.1.1.13)
116  #define OID_PKCS1_SM3_256_VALUE     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55,       \
117                                      0x01, 0x83, 0x78
118  SM3_256_OID(_PKCS1_);   // 1.2.156.10197.1.504
119  #define OID_PKCS1_SHA3_256_VALUE    NIST_SIG, 14
120  SHA3_256_OID(_PKCS1_);
121  #define OID_PKCS1_SHA3_384_VALUE    NIST_SIG, 15
122  SHA3_256_OID(_PKCS1_);
123  #define OID_PKCS1_SHA3_512_VALUE    NIST_SIG, 16
124  SHA3_512_OID(_PKCS1_);
125  #endif // ALG_RSA
126  #if ALG_ECDSA
127  #define OID_ECDSA_SHA1_VALUE        0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
128                                      0x01
129  SHA1_OID(_ECDSA_);      // (1.2.840.10045.4.1) SHA1 digest signed by an ECDSA key.
130  #define OID_ECDSA_SHA256_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
131                                      0x03, 0x02
132  SHA256_OID(_ECDSA_);    // (1.2.840.10045.4.3.2) SHA256 digest signed by an ECDSA key.
133  #define OID_ECDSA_SHA384_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
134                                      0x03, 0x03
135  SHA384_OID(_ECDSA_);    // (1.2.840.10045.4.3.3) SHA384 digest signed by an ECDSA key.
136  #define OID_ECDSA_SHA512_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
137                                      0x03, 0x04
138  SHA512_OID(_ECDSA_);    // (1.2.840.10045.4.3.4) SHA512 digest signed by an ECDSA key.
139  #define OID_ECDSA_SM3_256_VALUE     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01,  \
140                                      0x83, 0x75
141  SM3_256_OID(_ECDSA_);   // 1.2.156.10197.1.501
142  #define OID_ECDSA_SHA3_256_VALUE    NIST_SIG, 10
143  SHA3_256_OID(_ECDSA_);
144  #define OID_ECDSA_SHA3_384_VALUE    NIST_SIG, 11
145  SHA3_384_OID(_ECDSA_);
146  #define OID_ECDSA_SHA3_512_VALUE    NIST_SIG, 12
147  SHA3_512_OID(_ECDSA_);
148  #endif // ALG_ECDSA
149  #if ALG_ECC
150  #define OID_ECC_PUBLIC_VALUE        0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x02, \
151                                      0x01
152  MAKE_OID(_ECC_PUBLIC);
153  #define OID_ECC_NIST_P192_VALUE     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
154                                      0x01, 0x01
155  #if ECC_NIST_P192
156  MAKE_OID(_ECC_NIST_P192);   // (1.2.840.10045.3.1.1) 'nistP192'
157  #endif // ECC_NIST_P192
158  #define OID_ECC_NIST_P224_VALUE     0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x21
159  #if ECC_NIST_P224
160  MAKE_OID(_ECC_NIST_P224);   // (1.3.132.0.33)        'nistP224'
161  #endif // ECC_NIST_P224
162  #define OID_ECC_NIST_P256_VALUE     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
163                                      0x01, 0x07
164  #if ECC_NIST_P256
165  MAKE_OID(_ECC_NIST_P256);   // (1.2.840.10045.3.1.7)  'nistP256'
```

```
166   #endif // ECC_NIST_P256
167   #define OID_ECC_NIST_P384_VALUE     0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x22
168   #if ECC_NIST_P384
169   MAKE_OID(_ECC_NIST_P384);   // (1.3.132.0.34)         'nistP384'
170   #endif // ECC_NIST_P384
171   #define OID_ECC_NIST_P521_VALUE     0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x23
172   #if ECC_NIST_P521
173   MAKE_OID(_ECC_NIST_P521);   // (1.3.132.0.35)         'nistP521'
174   #endif // ECC_NIST_P521
```

No OIDs defined for these anonymous curves

```
175   #define OID_ECC_BN_P256_VALUE       0x00
176   #if ECC_BN_P256
177   MAKE_OID(_ECC_BN_P256);
178   #endif // ECC_BN_P256
179   #define OID_ECC_BN_P638_VALUE       0x00
180   #if ECC_BN_P638
181   MAKE_OID(_ECC_BN_P638);
182   #endif // ECC_BN_P638
183   #define OID_ECC_SM2_P256_VALUE      0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
184                                     0x82, 0x2D
185   #if ECC_SM2_P256
186   MAKE_OID(_ECC_SM2_P256);    // Don't know where I found this OID. It needs checking
187   #endif // ECC_SM2_P256
188   #if ECC_BN_P256
189   #define OID_ECC_BN_P256     NULL
190   #endif // ECC_BN_P256
191   #endif // ALG_ECC
192   #define OID_SIZE(OID)   (OID[1] + 2)
193   #endif // !_OIDS_H_
```

10.1.17 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 the state. The state is the size of the key plus the IV. The IV is a block. If Key = 256 and Block = 128 then 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_KEY_SIZE_BITS == 128
```

```
#   define DRBG_TEST_INITIATE_ENTROPY                  \
        0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
        0x93, 0xb6, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
        0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, \
        0x09, 0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
#   define DRBG_TEST_RESEED_ENTROPY                    \
        0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
        0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
        0x57, 0x4f, 0xc6, 0xdd, 0xd8, 0xbe, 0xd8, 0xbd, \
        0x72, 0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0x2a
#define DRBG_TEST_GENERATED_INTERM                    \
        0xdc, 0x3c, 0xf6, 0xbf, 0x5b, 0xd3, 0x41, 0x13, \
        0x5f, 0x2c, 0x68, 0x11, 0xa1, 0x07, 0x1c, 0x87
#   define DRBG_TEST_GENERATED                         \
        0xb6, 0x18, 0x50, 0xde, 0xcf, 0xd7, 0x10, 0x6d, \
        0x44, 0x76, 0x9a, 0x8e, 0x6e, 0x8c, 0x1a, 0xd4
#endif
#endif      //     _MSBN_DRBG_TEST_VECTORS_H
```

10.1.18 TpmAsn1.h

10.1.18.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
#ifndef _TPMASN1_H_
#define _TPMASN1_H_
```

10.1.18.2 Includes

```
#include "Tpm.h"
#include "OIDs.h"
```

10.1.18.3 Defined Constants

10.1.18.3.1 ASN.1 Universal Types (Class 00b

```
#define ASN1_EOC                0x00
#define ASN1_BOOLEAN            0x01
#define ASN1_INTEGER            0x02
#define ASN1_BITSTRING          0x03
#define ASN1_OCTET_STRING       0x04
#define ASN1_NULL               0x05
#define ASN1_OBJECT_IDENTIFIER  0x06
#define ASN1_OBJECT_DESCRIPTOR  0x07
#define ASN1_EXTERNAL           0x08
#define ASN1_REAL               0x09
#define ASN1_ENUMERATED         0x0A
#define ASN1_EMBEDDED           0x0B
#define ASN1_UTF8String         0x0C
#define ASN1_RELATIVE_OID       0x0D
#define ASN1_SEQUENCE           0x10        // Primitive + Constructed + 0x10
#define ASN1_SET                0x11        // Primitive + Constructed + 0x11
#define ASN1_NumericString      0x12
#define ASN1_PrintableString    0x13
#define ASN1_T61String          0x14
#define ASN1_VideoString        0x15
#define ASN1_IA5String          0x16
#define ASN1_UTCTime            0x17
#define ASN1_GeneralizeTime     0x18
#define ASN1_VisibleString      0x1A
```

```
29  #define ASN1_GeneralString      0x1B
30  #define ASN1_UniversalString    0x1C
31  #define ASN1_CHARACTER_STRING   0x1D
32  #define ASN1_BMPString          0x1E
33  #define ASN1_CONSTRUCTED        0x20
34  #define ASN1_APPLICAIION_SPECIFIC   0xA0
35  #define ASN1_CONSTRUCTED_SEQUENCE (ASN1_SEQUENCE + ASN1_CONSTRUCTED)
36  #define MAX_DEPTH              10  // maximum push depth for marshaling context.
```

10.1.18.4 Macros

10.1.18.4.1 Unmarshaling Macros

```
37  #ifndef VERIFY
38  #define VERIFY(_X_) {if(!(_X_)) goto Error; }
39  #endif
```

Checks the validity of the size making sure that there is no wrap around

```
40  #define CHECK_SIZE(context, length)                                          \
41              VERIFY(   (((length) + (context)->offset) >= (context)->offset)  \
42                     && (((length) + (context)->offset) <= (context)->size))
43  #define NEXT_OCTET(context) ((context)->buffer[(context)->offset++])
44  #define PEEK_NEXT(context) ((context)->buffer[(context)->offset])
```

10.1.18.4.2 Marshaling Macros

Marshaling works in reverse order. The offset is set to the top of the buffer and, as the buffer is filled, *offset* counts down to zero. When the full thing is encoded it can be moved to the top of the buffer. This happens when the last context is closed.

```
45  #define CHECK_SPACE(context, length)    VERIFY(context->offset > length)
```

10.1.18.5 Structures

```
46  typedef struct ASN1UnmarshalContext {
47      BYTE            *buffer;    // pointer to the buffer
48      INT16            size;      // size of the buffer (a negative number indicates
49                                  // a parsing failure).
50      INT16            offset;    // current offset into the buffer (a negative number
51                                  // indicates a parsing failure). Not used
52      BYTE             tag;       // The last unmarshaled tag
53  } ASN1UnmarshalContext;
54  typedef struct ASN1MarshalContext {
55      BYTE            *buffer;    // pointer to the start of the buffer
56      INT16            offset;    // place on the top where the last entry was added
57                                  // items are added from the bottom up.
58      INT16            end;       // the end offset of the current value
59      INT16            depth;     // how many pushed end values.
60      INT16            ends[MAX_DEPTH];
61  } ASN1MarshalContext;
62  #endif // _TPMASN1_H_
```

10.1.19 X509.h

10.1.19.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
#ifndef _X509_H_
#define _X509_H_
```

10.1.19.2 Includes

```
#include "Tpm.h"
#include "TpmASN1.h"
```

10.1.19.3 Defined Constants

10.1.19.3.1 X509 Application-specific types

```
#define X509_SELECTION          0xA0
#define X509_ISSUER_UNIQUE_ID   0xA1
#define X509_SUBJECT_UNIQUE_ID  0xA2
#define X509_EXTENSIONS         0xA3
```

These defines give the order in which values appear in the TBScertificate of an x.509 certificate. These values are used to index into an array of

```
#define ENCODED_SIZE_REF        0
#define VERSION_REF             (ENCODED_SIZE_REF + 1)
#define SERIAL_NUMBER_REF       (VERSION_REF + 1)
#define SIGNATURE_REF           (SERIAL_NUMBER_REF + 1)
#define ISSUER_REF              (SIGNATURE_REF + 1)
#define VALIDITY_REF            (ISSUER_REF + 1)
#define SUBJECT_KEY_REF         (VALIDITY_REF + 1)
#define SUBJECT_PUBLIC_KEY_REF  (SUBJECT_KEY_REF + 1)
#define EXTENSIONS_REF          (SUBJECT_PUBLIC_KEY_REF + 1)
#define REF_COUNT               (EXTENSIONS_REF + 1)
```

10.1.19.4 Structures

Used to access the fields of a TBSsignature some of which are in the *in_CertifyX509* structure and some of which are in the *out_CertifyX509* structure.

```
typedef struct stringRef
{
    BYTE        *buf;
    INT16        len;
} stringRef;
```

This is defined to avoid bit by bit comparisons within a UINT32

```
typedef union x509KeyUsageUnion {
    TPMA_X509_KEY_USAGE     x509;
    UINT32                  integer;
} x509KeyUsageUnion;
```

10.1.19.5 Global X509 Constants

These values are instanced by X509_spt.c and referenced by other X509-related files. This is the DER-encoded value for the Key Usage OID (2.5.29.15). This is the full OID, not just the numeric value

```
#define OID_KEY_USAGE_EXTENSION_VALUE  0x06, 0x03, 0x55, 0x1D, 0x0F
MAKE_OID(_KEY_USAGE_EXTENSION);
```

This is the DER-encoded value for the TCG-defined TPMA_OBJECT OID (2.23.133.10.1.1.1)

```
#define OID_TCG_TPMA_OBJECT_VALUE       0x06, 0x07, 0x67, 0x81, 0x05, 0x0a, 0x01,   \
                                        0x01, 0x01
MAKE_OID(_TCG_TPMA_OBJECT);
#ifdef _X509_SPT_
```

If a bit is SET in KEY_USAGE_SIGN is also SET in *keyUsagem* then the associated key has to have *sign* SET.

```
const x509KeyUsageUnion KEY_USAGE_SIGN =
{ TPMA_X509_KEY_USAGE_INITIALIZER(
    /* bits_at_0        */ 0, /* decipheronly    */ 0,  /* encipheronly   */ 0,
    /* crlsign          */ 1, /* keycertsign     */ 1,  /* keyagreement   */ 0,
    /* dataencipherment */ 0, /* keyencipherment */ 0,  /* nonrepudiation */ 0,
    /* digitalsignature */ 1) };
```

If a bit is SET in KEY_USAGE_DECRYPT is also SET in *keyUsagem* then the associated key has to have *decrypt* SET.

```
const x509KeyUsageUnion KEY_USAGE_DECRYPT =
{ TPMA_X509_KEY_USAGE_INITIALIZER(
    /* bits_at_0        */ 0, /* decipheronly    */ 1,  /* encipheronly   */ 1,
    /* crlsign          */ 0, /* keycertsign     */ 0,  /* keyagreement   */ 1,
    /* dataencipherment */ 1, /* keyencipherment */ 1,  /* nonrepudiation */ 0,
    /* digitalsignature */ 0) };
#else
extern x509KeyUsageUnion KEY_USAGE_SIGN;
extern x509KeyUsageUnion KEY_USAGE_DECRYPT;
#endif

#endif // _X509_H_
```

10.1.20 TpmAlgorithmDefines.h

This file contains the algorithm values from the TCG Algorithm Registry.

```
#ifndef _TPM_ALGORITHM_DEFINES_H_
#define _TPM_ALGORITHM_DEFINES_H_
```

Table 2:3 - Definition of Base Types Base Types are in BaseTypes.h

```
#define ECC_CURVES                          \
            {TPM_ECC_BN_P256,   TPM_ECC_BN_P638,   TPM_ECC_NIST_P192,             \
             TPM_ECC_NIST_P224, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384,             \
             TPM_ECC_NIST_P521, TPM_ECC_SM2_P256}
#define ECC_CURVE_COUNT                     \
            (ECC_BN_P256   + ECC_BN_P638   + ECC_NIST_P192 + ECC_NIST_P224 +      \
             ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 + ECC_SM2_P256)
#define MAX_ECC_KEY_BITS                    \
            MAX(ECC_BN_P256 * 256,   MAX(ECC_BN_P638 * 638,                       \
            MAX(ECC_NIST_P192 * 192, MAX(ECC_NIST_P224 * 224,                     \
            MAX(ECC_NIST_P256 * 256, MAX(ECC_NIST_P384 * 384,                     \
            MAX(ECC_NIST_P521 * 521, MAX(ECC_SM2_P256 * 256,                      \
            0))))))))
#define MAX_ECC_KEY_BYTES               BITS_TO_BYTES(MAX_ECC_KEY_BITS)
```

Table 0:6 - Defines for PLATFORM Values

```
#define PLATFORM_FAMILY         TPM_SPEC_FAMILY
#define PLATFORM_LEVEL          TPM_SPEC_LEVEL
#define PLATFORM_VERSION        TPM_SPEC_VERSION
#define PLATFORM_YEAR           TPM_SPEC_YEAR
#define PLATFORM_DAY_OF_YEAR    TPM_SPEC_DAY_OF_YEAR
```

Table 1:3 - Defines for RSA Asymmetric Cipher Algorithm Constants

```
#define RSA_KEY_SIZES_BITS          \
            (1024 * RSA_1024),  (2048 * RSA_2048), (3072 * RSA_3072),          \
             (4096 * RSA_4096)
#if   RSA_4096
#   define RSA_MAX_KEY_SIZE_BITS    4096
#elif RSA_3072
#   define RSA_MAX_KEY_SIZE_BITS    3072
#elif RSA_2048
#   define RSA_MAX_KEY_SIZE_BITS    2048
#elif RSA_1024
#   define RSA_MAX_KEY_SIZE_BITS    1024
#else
#   define RSA_MAX_KEY_SIZE_BITS    0
#endif
#define MAX_RSA_KEY_BITS            RSA_MAX_KEY_SIZE_BITS
#define MAX_RSA_KEY_BYTES           ((RSA_MAX_KEY_SIZE_BITS + 7) / 8)
```

Table 1:13 - Defines for SHA1 Hash Values

```
#define SHA1_DIGEST_SIZE    20
#define SHA1_BLOCK_SIZE     64
```

Table 1:14 - Defines for SHA256 Hash Values

```
#define SHA256_DIGEST_SIZE  32
#define SHA256_BLOCK_SIZE   64
```

Table 1:15 - Defines for SHA384 Hash Values

```
#define SHA384_DIGEST_SIZE  48
#define SHA384_BLOCK_SIZE   128
```

Table 1:16 - Defines for SHA512 Hash Values

```
#define SHA512_DIGEST_SIZE  64
#define SHA512_BLOCK_SIZE   128
```

Table 1:17 - Defines for SM3_256 Hash Values

```
#define SM3_256_DIGEST_SIZE     32
#define SM3_256_BLOCK_SIZE      64
```

Table 1:18 - Defines for SHA3_256 Hash Values

```
#define SHA3_256_DIGEST_SIZE    32
#define SHA3_256_BLOCK_SIZE     136
```

Table 1:19 - Defines for SHA3_384 Hash Values

```
#define SHA3_384_DIGEST_SIZE    48
#define SHA3_384_BLOCK_SIZE     104
```

Table 1:20 - Defines for SHA3_512 Hash Values

```
#define SHA3_512_DIGEST_SIZE    64
#define SHA3_512_BLOCK_SIZE     72
```

Table 1:21 - Defines for AES Symmetric Cipher Algorithm Constants

```
#define AES_KEY_SIZES_BITS          \
            (128 * AES_128), (192 * AES_192), (256 * AES_256)
#if   AES_256
#   define AES_MAX_KEY_SIZE_BITS    256
#elif AES_192
#   define AES_MAX_KEY_SIZE_BITS    192
#elif AES_128
#   define AES_MAX_KEY_SIZE_BITS    128
#else
#   define AES_MAX_KEY_SIZE_BITS    0
#endif
#define MAX_AES_KEY_BITS            AES_MAX_KEY_SIZE_BITS
#define MAX_AES_KEY_BYTES           ((AES_MAX_KEY_SIZE_BITS + 7) / 8)
#define AES_128_BLOCK_SIZE_BYTES    (AES_128 * 16)
#define AES_192_BLOCK_SIZE_BYTES    (AES_192 * 16)
#define AES_256_BLOCK_SIZE_BYTES    (AES_256 * 16)
#define AES_BLOCK_SIZES             \
            AES_128_BLOCK_SIZE_BYTES, AES_192_BLOCK_SIZE_BYTES,                    \
            AES_256_BLOCK_SIZE_BYTES
#if   ALG_AES
#   define AES_MAX_BLOCK_SIZE       16
#else
#   define AES_MAX_BLOCK_SIZE       0
#endif
#define MAX_AES_BLOCK_SIZE_BYTES    AES_MAX_BLOCK_SIZE
```

Table 1:22 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
#define SM4_KEY_SIZES_BITS          (128 * SM4_128)
#if   SM4_128
#   define SM4_MAX_KEY_SIZE_BITS    128
#else
#   define SM4_MAX_KEY_SIZE_BITS    0
#endif
#define MAX_SM4_KEY_BITS            SM4_MAX_KEY_SIZE_BITS
#define MAX_SM4_KEY_BYTES           ((SM4_MAX_KEY_SIZE_BITS + 7) / 8)
#define SM4_128_BLOCK_SIZE_BYTES    (SM4_128 * 16)
#define SM4_BLOCK_SIZES             SM4_128_BLOCK_SIZE_BYTES
#if   ALG_SM4
#   define SM4_MAX_BLOCK_SIZE       16
#else
#   define SM4_MAX_BLOCK_SIZE       0
#endif
#define MAX_SM4_BLOCK_SIZE_BYTES    SM4_MAX_BLOCK_SIZE
```

Table 1:23 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
#define CAMELLIA_KEY_SIZES_BITS         \
            (128 * CAMELLIA_128), (192 * CAMELLIA_192), (256 * CAMELLIA_256)
#if   CAMELLIA_256
#   define CAMELLIA_MAX_KEY_SIZE_BITS   256
#elif CAMELLIA_192
#   define CAMELLIA_MAX_KEY_SIZE_BITS   192
#elif CAMELLIA_128
#   define CAMELLIA_MAX_KEY_SIZE_BITS   128
#else
#   define CAMELLIA_MAX_KEY_SIZE_BITS   0
#endif
#define MAX_CAMELLIA_KEY_BITS           CAMELLIA_MAX_KEY_SIZE_BITS
#define MAX_CAMELLIA_KEY_BYTES          ((CAMELLIA_MAX_KEY_SIZE_BITS + 7) / 8)
#define CAMELLIA_128_BLOCK_SIZE_BYTES   (CAMELLIA_128 * 16)
#define CAMELLIA_192_BLOCK_SIZE_BYTES   (CAMELLIA_192 * 16)
#define CAMELLIA_256_BLOCK_SIZE_BYTES   (CAMELLIA_256 * 16)
#define CAMELLIA_BLOCK_SIZES            \
            CAMELLIA_128_BLOCK_SIZE_BYTES, CAMELLIA_192_BLOCK_SIZE_BYTES,          \
```

```
113             CAMELLIA_256_BLOCK_SIZE_BYTES
114     #if   ALG_CAMELLIA
115     #   define CAMELLIA_MAX_BLOCK_SIZE      16
116     #else
117     #   define CAMELLIA_MAX_BLOCK_SIZE      0
118     #endif
119     #define MAX_CAMELLIA_BLOCK_SIZE_BYTES   CAMELLIA_MAX_BLOCK_SIZE
```

Table 1:24 - Defines for TDES Symmetric Cipher Algorithm Constants

```
120     #define TDES_KEY_SIZES_BITS         (128 * TDES_128), (192 * TDES_192)
121     #if   TDES_192
122     #   define TDES_MAX_KEY_SIZE_BITS   192
123     #elif TDES_128
124     #   define TDES_MAX_KEY_SIZE_BITS   128
125     #else
126     #   define TDES_MAX_KEY_SIZE_BITS   0
127     #endif
128     #define MAX_TDES_KEY_BITS           TDES_MAX_KEY_SIZE_BITS
129     #define MAX_TDES_KEY_BYTES          ((TDES_MAX_KEY_SIZE_BITS + 7) / 8)
130     #define TDES_128_BLOCK_SIZE_BYTES   (TDES_128 * 8)
131     #define TDES_192_BLOCK_SIZE_BYTES   (TDES_192 * 8)
132     #define TDES_BLOCK_SIZES            \
133                 TDES_128_BLOCK_SIZE_BYTES, TDES_192_BLOCK_SIZE_BYTES
134     #if   ALG_TDES
135     #   define TDES_MAX_BLOCK_SIZE      8
136     #else
137     #   define TDES_MAX_BLOCK_SIZE      0
138     #endif
139     #define MAX_TDES_BLOCK_SIZE_BYTES   TDES_MAX_BLOCK_SIZE
```

Additional values for benefit of code

```
140     #define TPM_CC_FIRST                        0x0000011F
141     #define TPM_CC_LAST                         0x00000198
142     #if COMPRESSED_LISTS
143     #define ADD_FILL            0
144     #else
145     #define ADD_FILL            1
146     #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)

```
147     #define LIBRARY_COMMAND_ARRAY_SIZE       (0      \
148         + (ADD_FILL || CC_NV_UndefineSpaceSpecial)              /* 0x0000011F */       \
149         + (ADD_FILL || CC_EvictControl)                         /* 0x00000120 */       \
150         + (ADD_FILL || CC_HierarchyControl)                     /* 0x00000121 */       \
151         + (ADD_FILL || CC_NV_UndefineSpace)                     /* 0x00000122 */       \
152         +  ADD_FILL                                             /* 0x00000123 */       \
153         + (ADD_FILL || CC_ChangeEPS)                            /* 0x00000124 */       \
154         + (ADD_FILL || CC_ChangePPS)                            /* 0x00000125 */       \
155         + (ADD_FILL || CC_Clear)                                /* 0x00000126 */       \
156         + (ADD_FILL || CC_ClearControl)                         /* 0x00000127 */       \
157         + (ADD_FILL || CC_ClockSet)                             /* 0x00000128 */       \
158         + (ADD_FILL || CC_HierarchyChangeAuth)                  /* 0x00000129 */       \
159         + (ADD_FILL || CC_NV_DefineSpace)                       /* 0x0000012A */       \
160         + (ADD_FILL || CC_PCR_Allocate)                         /* 0x0000012B */       \
161         + (ADD_FILL || CC_PCR_SetAuthPolicy)                    /* 0x0000012C */       \
162         + (ADD_FILL || CC_PP_Commands)                          /* 0x0000012D */       \
163         + (ADD_FILL || CC_SetPrimaryPolicy)                     /* 0x0000012E */       \
164         + (ADD_FILL || CC_FieldUpgradeStart)                    /* 0x0000012F */       \
165         + (ADD_FILL || CC_ClockRateAdjust)                      /* 0x00000130 */       \
166         + (ADD_FILL || CC_CreatePrimary)                        /* 0x00000131 */       \
```

```
167       + (ADD_FILL || CC_NV_GlobalWriteLock)                 /* 0x00000132 */       \
168       + (ADD_FILL || CC_GetCommandAuditDigest)              /* 0x00000133 */       \
169       + (ADD_FILL || CC_NV_Increment)                       /* 0x00000134 */       \
170       + (ADD_FILL || CC_NV_SetBits)                         /* 0x00000135 */       \
171       + (ADD_FILL || CC_NV_Extend)                          /* 0x00000136 */       \
172       + (ADD_FILL || CC_NV_Write)                           /* 0x00000137 */       \
173       + (ADD_FILL || CC_NV_WriteLock)                       /* 0x00000138 */       \
174       + (ADD_FILL || CC_DictionaryAttackLockReset)          /* 0x00000139 */       \
175       + (ADD_FILL || CC_DictionaryAttackParameters)         /* 0x0000013A */       \
176       + (ADD_FILL || CC_NV_ChangeAuth)                      /* 0x0000013B */       \
177       + (ADD_FILL || CC_PCR_Event)                          /* 0x0000013C */       \
178       + (ADD_FILL || CC_PCR_Reset)                          /* 0x0000013D */       \
179       + (ADD_FILL || CC_SequenceComplete)                   /* 0x0000013E */       \
180       + (ADD_FILL || CC_SetAlgorithmSet)                    /* 0x0000013F */       \
181       + (ADD_FILL || CC_SetCommandCodeAuditStatus)          /* 0x00000140 */       \
182       + (ADD_FILL || CC_FieldUpgradeData)                   /* 0x00000141 */       \
183       + (ADD_FILL || CC_IncrementalSelfTest)                /* 0x00000142 */       \
184       + (ADD_FILL || CC_SelfTest)                           /* 0x00000143 */       \
185       + (ADD_FILL || CC_Startup)                            /* 0x00000144 */       \
186       + (ADD_FILL || CC_Shutdown)                           /* 0x00000145 */       \
187       + (ADD_FILL || CC_StirRandom)                         /* 0x00000146 */       \
188       + (ADD_FILL || CC_ActivateCredential)                 /* 0x00000147 */       \
189       + (ADD_FILL || CC_Certify)                            /* 0x00000148 */       \
190       + (ADD_FILL || CC_PolicyNV)                           /* 0x00000149 */       \
191       + (ADD_FILL || CC_CertifyCreation)                    /* 0x0000014A */       \
192       + (ADD_FILL || CC_Duplicate)                          /* 0x0000014B */       \
193       + (ADD_FILL || CC_GetTime)                            /* 0x0000014C */       \
194       + (ADD_FILL || CC_GetSessionAuditDigest)              /* 0x0000014D */       \
195       + (ADD_FILL || CC_NV_Read)                            /* 0x0000014E */       \
196       + (ADD_FILL || CC_NV_ReadLock)                        /* 0x0000014F */       \
197       + (ADD_FILL || CC_ObjectChangeAuth)                   /* 0x00000150 */       \
198       + (ADD_FILL || CC_PolicySecret)                       /* 0x00000151 */       \
199       + (ADD_FILL || CC_Rewrap)                             /* 0x00000152 */       \
200       + (ADD_FILL || CC_Create)                             /* 0x00000153 */       \
201       + (ADD_FILL || CC_ECDH_ZGen)                          /* 0x00000154 */       \
202       + (ADD_FILL || CC_HMAC || CC_MAC)                     /* 0x00000155 */       \
203       + (ADD_FILL || CC_Import)                             /* 0x00000156 */       \
204       + (ADD_FILL || CC_Load)                               /* 0x00000157 */       \
205       + (ADD_FILL || CC_Quote)                              /* 0x00000158 */       \
206       + (ADD_FILL || CC_RSA_Decrypt)                        /* 0x00000159 */       \
207       +  ADD_FILL                                           /* 0x0000015A */       \
208       + (ADD_FILL || CC_HMAC_Start || CC_MAC_Start)         /* 0x0000015B */       \
209       + (ADD_FILL || CC_SequenceUpdate)                     /* 0x0000015C */       \
210       + (ADD_FILL || CC_Sign)                               /* 0x0000015D */       \
211       + (ADD_FILL || CC_Unseal)                             /* 0x0000015E */       \
212       +  ADD_FILL                                           /* 0x0000015F */       \
213       + (ADD_FILL || CC_PolicySigned)                       /* 0x00000160 */       \
214       + (ADD_FILL || CC_ContextLoad)                        /* 0x00000161 */       \
215       + (ADD_FILL || CC_ContextSave)                        /* 0x00000162 */       \
216       + (ADD_FILL || CC_ECDH_KeyGen)                        /* 0x00000163 */       \
217       + (ADD_FILL || CC_EncryptDecrypt)                     /* 0x00000164 */       \
218       + (ADD_FILL || CC_FlushContext)                       /* 0x00000165 */       \
219       +  ADD_FILL                                           /* 0x00000166 */       \
220       + (ADD_FILL || CC_LoadExternal)                       /* 0x00000167 */       \
221       + (ADD_FILL || CC_MakeCredential)                     /* 0x00000168 */       \
222       + (ADD_FILL || CC_NV_ReadPublic)                      /* 0x00000169 */       \
223       + (ADD_FILL || CC_PolicyAuthorize)                    /* 0x0000016A */       \
224       + (ADD_FILL || CC_PolicyAuthValue)                    /* 0x0000016B */       \
225       + (ADD_FILL || CC_PolicyCommandCode)                  /* 0x0000016C */       \
226       + (ADD_FILL || CC_PolicyCounterTimer)                 /* 0x0000016D */       \
227       + (ADD_FILL || CC_PolicyCpHash)                       /* 0x0000016E */       \
228       + (ADD_FILL || CC_PolicyLocality)                     /* 0x0000016F */       \
229       + (ADD_FILL || CC_PolicyNameHash)                     /* 0x00000170 */       \
230       + (ADD_FILL || CC_PolicyOR)                           /* 0x00000171 */       \
231       + (ADD_FILL || CC_PolicyTicket)                       /* 0x00000172 */       \
232       + (ADD_FILL || CC_ReadPublic)                         /* 0x00000173 */       \
```

```
233         + (ADD_FILL || CC_RSA_Encrypt)                          /* 0x00000174 */        \
234         +  ADD_FILL                                             /* 0x00000175 */        \
235         + (ADD_FILL || CC_StartAuthSession)                     /* 0x00000176 */        \
236         + (ADD_FILL || CC_VerifySignature)                      /* 0x00000177 */        \
237         + (ADD_FILL || CC_ECC_Parameters)                       /* 0x00000178 */        \
238         + (ADD_FILL || CC_FirmwareRead)                         /* 0x00000179 */        \
239         + (ADD_FILL || CC_GetCapability)                        /* 0x0000017A */        \
240         + (ADD_FILL || CC_GetRandom)                            /* 0x0000017B */        \
241         + (ADD_FILL || CC_GetTestResult)                        /* 0x0000017C */        \
242         + (ADD_FILL || CC_Hash)                                 /* 0x0000017D */        \
243         + (ADD_FILL || CC_PCR_Read)                             /* 0x0000017E */        \
244         + (ADD_FILL || CC_PolicyPCR)                            /* 0x0000017F */        \
245         + (ADD_FILL || CC_PolicyRestart)                        /* 0x00000180 */        \
246         + (ADD_FILL || CC_ReadClock)                            /* 0x00000181 */        \
247         + (ADD_FILL || CC_PCR_Extend)                           /* 0x00000182 */        \
248         + (ADD_FILL || CC_PCR_SetAuthValue)                     /* 0x00000183 */        \
249         + (ADD_FILL || CC_NV_Certify)                           /* 0x00000184 */        \
250         + (ADD_FILL || CC_EventSequenceComplete)                /* 0x00000185 */        \
251         + (ADD_FILL || CC_HashSequenceStart)                    /* 0x00000186 */        \
252         + (ADD_FILL || CC_PolicyPhysicalPresence)               /* 0x00000187 */        \
253         + (ADD_FILL || CC_PolicyDuplicationSelect)              /* 0x00000188 */        \
254         + (ADD_FILL || CC_PolicyGetDigest)                      /* 0x00000189 */        \
255         + (ADD_FILL || CC_TestParms)                            /* 0x0000018A */        \
256         + (ADD_FILL || CC_Commit)                               /* 0x0000018B */        \
257         + (ADD_FILL || CC_PolicyPassword)                       /* 0x0000018C */        \
258         + (ADD_FILL || CC_ZGen_2Phase)                          /* 0x0000018D */        \
259         + (ADD_FILL || CC_EC_Ephemeral)                         /* 0x0000018E */        \
260         + (ADD_FILL || CC_PolicyNvWritten)                      /* 0x0000018F */        \
261         + (ADD_FILL || CC_PolicyTemplate)                       /* 0x00000190 */        \
262         + (ADD_FILL || CC_CreateLoaded)                         /* 0x00000191 */        \
263         + (ADD_FILL || CC_PolicyAuthorizeNV)                    /* 0x00000192 */        \
264         + (ADD_FILL || CC_EncryptDecrypt2)                      /* 0x00000193 */        \
265         + (ADD_FILL || CC_AC_GetCapability)                     /* 0x00000194 */        \
266         + (ADD_FILL || CC_AC_Send)                              /* 0x00000195 */        \
267         + (ADD_FILL || CC_Policy_AC_SendSelect)                 /* 0x00000196 */        \
268         + (ADD_FILL || CC_CertifyX509)                          /* 0x00000197 */        \
269         + (ADD_FILL || CC_ACT_SetTimeout)                       /* 0x00000198 */        \
270         )
271     #define VENDOR_COMMAND_ARRAY_SIZE   (0 + CC_Vendor_TCG_Test)
272     #define COMMAND_COUNT       (LIBRARY_COMMAND_ARRAY_SIZE + VENDOR_COMMAND_ARRAY_SIZE)
273     #define HASH_COUNT          \
274                 (ALG_SHA1     + ALG_SHA256   + ALG_SHA384   + ALG_SHA3_256 +          \
275                  ALG_SHA3_384 + ALG_SHA3_512 + ALG_SHA512   + ALG_SM3_256)
276     #define MAX_HASH_BLOCK_SIZE \
277                 (MAX(ALG_SHA1     * SHA1_BLOCK_SIZE,                                    \
278                  MAX(ALG_SHA256   * SHA256_BLOCK_SIZE,                                  \
279                  MAX(ALG_SHA384   * SHA384_BLOCK_SIZE,                                  \
280                  MAX(ALG_SHA3_256 * SHA3_256_BLOCK_SIZE,                                \
281                  MAX(ALG_SHA3_384 * SHA3_384_BLOCK_SIZE,                                \
282                  MAX(ALG_SHA3_512 * SHA3_512_BLOCK_SIZE,                                \
283                  MAX(ALG_SHA512   * SHA512_BLOCK_SIZE,                                  \
284                  MAX(ALG_SM3_256  * SM3_256_BLOCK_SIZE,                                 \
285                  0))))))))))
286     #define MAX_DIGEST_SIZE     \
287                 (MAX(ALG_SHA1     * SHA1_DIGEST_SIZE,                                   \
288                  MAX(ALG_SHA256   * SHA256_DIGEST_SIZE,                                 \
289                  MAX(ALG_SHA384   * SHA384_DIGEST_SIZE,                                 \
290                  MAX(ALG_SHA3_256 * SHA3_256_DIGEST_SIZE,                               \
291                  MAX(ALG_SHA3_384 * SHA3_384_DIGEST_SIZE,                               \
292                  MAX(ALG_SHA3_512 * SHA3_512_DIGEST_SIZE,                               \
293                  MAX(ALG_SHA512   * SHA512_DIGEST_SIZE,                                 \
294                  MAX(ALG_SM3_256  * SM3_256_DIGEST_SIZE,                                \
295                  0))))))))))
296     #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
297     #error "Hash data not valid"
298     #endif
```

Define the 2B structure that would hold any hash block

```
299  TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

Following typedef is for some old code

```
300  typedef TPM2B_MAX_HASH_BLOCK    TPM2B_HASH_BLOCK;
```

Additional symmetric constants

```
301  #define MAX_SYM_KEY_BITS        \
302              (MAX(AES_MAX_KEY_SIZE_BITS,      MAX(CAMELLIA_MAX_KEY_SIZE_BITS,       \
303               MAX(SM4_MAX_KEY_SIZE_BITS,      MAX(TDES_MAX_KEY_SIZE_BITS,           \
304               0)))))
305  #define MAX_SYM_KEY_BYTES       ((MAX_SYM_KEY_BITS + 7) / 8)
306  #define MAX_SYM_BLOCK_SIZE      \
307              (MAX(AES_MAX_BLOCK_SIZE,      MAX(CAMELLIA_MAX_BLOCK_SIZE,            \
308               MAX(SM4_MAX_BLOCK_SIZE,      MAX(TDES_MAX_BLOCK_SIZE,                \
309               0)))))
310  #if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
311  #   error Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK
312  #endif
313  #endif // _TPM_ALGORITHM_DEFINES_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

```
#include     "Tpm.h"
#define      SELF_TEST_DATA
#if SELF_TEST
```

These includes pull in the data structures. They contain data definitions for the various tests.

```
#include     "SelfTest.h"
#include     "SymmetricTest.h"
#include     "RsaTestData.h"
#include     "EccTestData.h"
#include     "HashTestData.h"
#include     "KdfTestData.h"
#define TEST_DEFAULT_TEST_HASH(vector)                                          \
            if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest))                           \
                TestHash(DEFAULT_TEST_HASH, vector);
```

Make sure that the algorithm has been tested

```
#define CLEAR_BOTH(alg)      {   CLEAR_BIT(alg, *toTest);                       \
                                 if(toTest != &g_toTest)                        \
                                     CLEAR_BIT(alg, g_toTest); }
#define SET_BOTH(alg)      {   SET_BIT(alg, *toTest);                           \
                                 if(toTest != &g_toTest)                        \
                                     SET_BIT(alg, g_toTest); }
#define TEST_BOTH(alg)       ((toTest != &g_toTest)                             \
                             ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest)\
                             : TEST_BIT(alg, *toTest))
```

Can only cancel if doing a list.

```
#define CHECK_CANCELED                                                          \
    if(_plat__IsCanceled() && toTest != &g_toTest)                              \
        return TPM_RC_CANCELED;
```

10.2.1.3 Hash Tests

10.2.1.3.1 Description

The hash test does a known-value HMAC using the specified hash algorithm.

10.2.1.3.2 TestHash()

The hash test function.

```
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  // There are currently not test vectors for SM3
63  //            testDigest = &c_SM3_256_digest.b;
64              testDigest = NULL;
65              break;
66  #endif
67          default:
68              FAIL(FATAL_ERROR_INTERNAL);
69      }
70      // Clear the to-test bits
71      CLEAR_BOTH(hashAlg);
72
73      // If there is an algorithm without test vectors, then assume that things are OK.
74      if(testDigest == NULL)
75          return TPM_RC_SUCCESS;
76
77      // Set the HMAC key to twice the digest size
78      digestSize = CryptHashGetDigestSize(hashAlg);
79      CryptHmacStart(&state, hashAlg, digestSize * 2,
80                     (BYTE *)c_hashTestKey.t.buffer);
81      CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
82                        (BYTE *)c_hashTestData.t.buffer);
83      computed.t.size = digestSize;
84      CryptHmacEnd(&state, digestSize, computed.t.buffer);
85      if((testDigest->size != computed.t.size)
86         || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
87          SELF_TEST_FAILURE;
88      return TPM_RC_SUCCESS;
89  }
```

10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```
90  static UINT32
91  MakeIv(
92      TPM_ALG_ID    mode,      // IN: symmetric mode
93      UINT32        size,      // IN: block size of the algorithm
94      BYTE         *iv         // OUT: IV to fill in
95      )
96  {
97      BYTE          i;
98
99      if(mode == ALG_ECB_VALUE)
100         return 0;
101     if(mode == ALG_CTR_VALUE)
102     {
103         // The test uses an IV that has 0xff in the last byte
104         for(i = 1; i <= size; i++)
105             *iv++ = 0xff - (BYTE)(size - i);
106     }
107     else
108     {
109         for(i = 0; i < size; i++)
110             *iv++ = i;
111     }
112     return size;
113 }
```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```
114 static void
115 TestSymmetricAlgorithm(
116     const SYMMETRIC_TEST_VECTOR     *test,          //
117     TPM_ALG_ID                       mode           //
118     )
119 {
120     static BYTE                 encrypted[MAX_SYM_BLOCK_SIZE * 2];
121     static BYTE                 decrypted[MAX_SYM_BLOCK_SIZE * 2];
122     static TPM2B_IV             iv;
123 //
124     // Get the appropriate IV
125     iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
126
127     // Encrypt known data
128     CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
129                           mode, test->dataInOutSize, test->dataIn);
130     // Check that it matches the expected value
131     if(!MemoryEqual(encrypted, test->dataOut[mode - ALG_CTR_VALUE],
132                     test->dataInOutSize))
133         SELF_TEST_FAILURE;
134     // Reinitialize the iv for decryption
135     MakeIv(mode, test->ivSize, iv.t.buffer);
136     CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
137                           mode, test->dataInOutSize,
138                           test->dataOut[mode - ALG_CTR_VALUE]);
139     // Make sure that it matches what we started with
140     if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
141         SELF_TEST_FAILURE;
```

```
142     }
```

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

```
143     static BOOL
144     AllSymsAreDone(
145         ALGORITHM_VECTOR        *toTest
146         )
147     {
148         return (!TEST_BOTH(ALG_AES_VALUE) && !TEST_BOTH(ALG_SM4_VALUE));
149     }
```

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

```
150     static BOOL
151     AllModesAreDone(
152         ALGORITHM_VECTOR            *toTest
153         )
154     {
155         TPM_ALG_ID                  alg;
156         for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
157             if(TEST_BOTH(alg))
158                 return FALSE;
159         return TRUE;
160     }
```

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.

```
161     static TPM_RC
162     TestSymmetric(
163         TPM_ALG_ID                   alg,
164         ALGORITHM_VECTOR            *toTest
165         )
166     {
167         SYM_INDEX                    index;
168         TPM_ALG_ID                   mode;
169     //
170         if(!TEST_BIT(alg, *toTest))
171             return TPM_RC_SUCCESS;
172         if(alg == ALG_AES_VALUE || alg == ALG_SM4_VALUE || alg == ALG_CAMELLIA_VALUE)
173         {
```

```
174         // Will test the algorithm for all modes and key sizes
175         CLEAR_BOTH(alg);
176 
177         // A test this algorithm for all modes
178         for(index = 0; index < NUM_SYMS; index++)
179         {
180             if(c_symTestValues[index].alg == alg)
181             {
182                 for(mode = TPM_SYM_MODE_FIRST;
183                 mode <= TPM_SYM_MODE_LAST;
184                     mode++)
185                 {
186                     if(TEST_BIT(mode, *toTest))
187                         TestSymmetricAlgorithm(&c_symTestValues[index], mode);
188                 }
189             }
190         }
191         // if all the symmetric tests are done
192         if(AllSymsAreDone(toTest))
193         {
194             // all symmetric algorithms tested so no modes should be set
195             for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
196                 CLEAR_BOTH(alg);
197         }
198     }
199     else if(TPM_SYM_MODE_FIRST <= alg && alg <= TPM_SYM_MODE_LAST)
200     {
201         // Test this mode for all key sizes and algorithms
202         for(index = 0; index < NUM_SYMS; index++)
203         {
204             // The mode testing only comes into play when doing self tests
205             // by command. When doing self tests by command, the block ciphers are
206             // tested first. That means that all of their modes would have been
207             // tested for all key sizes. If there is no block cipher left to
208             // test, then clear this mode bit.
209             if(!TEST_BIT(ALG_AES_VALUE, *toTest)
210                && !TEST_BIT(ALG_SM4_VALUE, *toTest))
211             {
212                 CLEAR_BOTH(alg);
213             }
214             else
215             {
216                 for(index = 0; index < NUM_SYMS; index++)
217                 {
218                     if(TEST_BIT(c_symTestValues[index].alg, *toTest))
219                         TestSymmetricAlgorithm(&c_symTestValues[index], alg);
220                 }
221                 // have tested this mode for all algorithms
222                 CLEAR_BOTH(alg);
223             }
224         }
225         if(AllModesAreDone(toTest))
226         {
227             CLEAR_BOTH(ALG_AES_VALUE);
228             CLEAR_BOTH(ALG_SM4_VALUE);
229         }
230     }
231     else
232         pAssert(alg == 0 && alg != 0);
233     return TPM_RC_SUCCESS;
234 }
```

10.2.1.5 RSA Tests

```
235 #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.

```
236 static void
237 RsaKeyInitialize(
238     OBJECT          *testObject
239     )
240 {
241     MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
242                  sizeof(c_rsaPublicModulus));
243     MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
244                  sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
245     testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
246     // Use the default exponent
247     testObject->publicArea.parameters.rsaDetail.exponent = 0;
248 }
```

10.2.1.5.3 TestRsaEncryptDecrypt()

These tests are for a public key encryption that uses a random value.

```
249 static TPM_RC
250 TestRsaEncryptDecrypt(
251     TPM_ALG_ID           scheme,            // IN: the scheme
252     ALGORITHM_VECTOR    *toTest             //
253     )
254 {
255     static TPM2B_PUBLIC_KEY_RSA      testInput;
256     static TPM2B_PUBLIC_KEY_RSA      testOutput;
257     static OBJECT                    testObject;
258     const TPM2B_RSA_TEST_KEY        *kvtValue = NULL;
259     TPM_RC                           result = TPM_RC_SUCCESS;
260     const TPM2B                     *testLabel = NULL;
261     TPMT_RSA_DECRYPT                 rsaScheme;
262 //
263     // Don't need to initialize much of the test object
264     RsaKeyInitialize(&testObject);
265     rsaScheme.scheme = scheme;
266     rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
267     CLEAR_BOTH(scheme);
268     CLEAR_BOTH(ALG_NULL_VALUE);
269     if(scheme == ALG_NULL_VALUE)
270     {
```

```
271             // This is an encryption scheme using the private key without any encoding.
272             memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
273             testInput.t.size = sizeof(c_RsaTestValue);
274             if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
275                                                  &testObject, &rsaScheme, NULL, NULL))
276                 SELF_TEST_FAILURE;
277             if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
278                 SELF_TEST_FAILURE;
279             MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
280             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
281                                                  &testObject, &rsaScheme, NULL))
282                 SELF_TEST_FAILURE;
283             if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
284                             sizeof(c_RsaTestValue)))
285                 SELF_TEST_FAILURE;
286         }
287         else
288         {
289             // ALG_RSAES_VALUE:
290             // This is an decryption scheme using padding according to
291             // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
292             // key encryption that uses random data, encrypt a value and then
293             // decrypt the value and see that we get the encrypted data back.
294             // The hash is not used by this encryption so it can be TMP_ALG_NULL
295 
296             // ALG_OAEP_VALUE:
297             // This is also an decryption scheme and it also uses a
298             // pseudo-random
299             // value. However, this also uses a hash algorithm. So, we may need
300             // to test that algorithm before use.
301             if(scheme == ALG_OAEP_VALUE)
302             {
303                 TEST_DEFAULT_TEST_HASH(toTest);
304                 kvtValue = &c_OaepKvt;
305                 testLabel = OAEP_TEST_STRING;
306             }
307             else if(scheme == ALG_RSAES_VALUE)
308             {
309                 kvtValue = &c_RsaesKvt;
310                 testLabel = NULL;
311             }
312             else
313                 SELF_TEST_FAILURE;
314             // Only use a digest-size portion of the test value
315             memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
316             testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
317 
318             // See if the encryption works
319             if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
320                                                  &testObject, &rsaScheme, testLabel,
321                                                  NULL))
322                 SELF_TEST_FAILURE;
323             MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
324             // see if we can decrypt this value and get the original data back
325             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
326                                                  &testObject, &rsaScheme, testLabel))
327                 SELF_TEST_FAILURE;
328             // See if the results compare
329             if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
330                || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
331                                DEFAULT_TEST_DIGEST_SIZE))
332                 SELF_TEST_FAILURE;
333             // Now check that the decryption works on a known value
334             MemoryCopy2B(&testInput.b, (P2B)kvtValue,
335                          sizeof(testInput.t.buffer));
336             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
```

```
                                                &testObject, &rsaScheme, testLabel))
            SELF_TEST_FAILURE;
        if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
           || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
                           DEFAULT_TEST_DIGEST_SIZE))
            SELF_TEST_FAILURE;
    }
    return result;
}
```

10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

```
static TPM_RC
TestRsaSignAndVerify(
    TPM_ALG_ID               scheme,
    ALGORITHM_VECTOR        *toTest
    )
{
    TPM_RC                      result = TPM_RC_SUCCESS;
    static OBJECT               testObject;
    static TPM2B_DIGEST         testDigest;
    static TPMT_SIGNATURE       testSig;

    // Do a sign and signature verification.
    // RSASSA:
    // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
    // use random data so there is a KVT for the signing operation. On
    // first use of the scheme for signing, use the TPM's RSA key to
    // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
    // decrypt the data to see that it matches the starting value. This verifies
    // the signature with a KVT

    // Clear the bits indicating that the function has not been checked. This is to
    // prevent looping
    CLEAR_BOTH(scheme);
    CLEAR_BOTH(ALG_NULL_VALUE);
    CLEAR_BOTH(ALG_RSA_VALUE);

    RsaKeyInitialize(&testObject);
    memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
    testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
    testSig.sigAlg = scheme;
    testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;

    // RSAPSS:
    // This is a signing scheme a according to PKCS#1-v2.2 8.1 it uses
    // random data in the signature so there is no KVT for the signing
    // operation. To test signing, the TPM will use the TPM's RSA key
    // to sign a portion of c_RsaTestValue and then it will verify the
    // signature. For verification, c_RsapssKvt is verified before the
    // user signature blob is verified. The worst case for testing of this
    // algorithm is two private and one public key operation.

    // The process is to sign known data. If RSASSA is being done, verify that the
    // signature matches the precomputed value. For both, use the signed value and
    // see that the verification says that it is a good signature. Then
    // if testing RSAPSS, do a verify of a known good signature. This ensures that
    // the validation function works.

    if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
        SELF_TEST_FAILURE;
    // For RSASSA, make sure the results is what we are looking for
```

```
396         if(testSig.sigAlg == ALG_RSASSA_VALUE)
397         {
398             if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
399                || !MemoryEqual(c_RsassaKvt.buffer,
400                                testSig.signature.rsassa.sig.t.buffer,
401                                RSA_TEST_KEY_SIZE))
402                 SELF_TEST_FAILURE;
403         }
404         // See if the TPM will validate its own signatures
405         if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
406                                                        &testDigest))
407             SELF_TEST_FAILURE;
408         // If this is RSAPSS, check the verification with known signature
409         // Have to copy because  CrytpRsaValidateSignature() eats the signature
410         if(ALG_RSAPSS_VALUE == scheme)
411         {
412             MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
413                          sizeof(testSig.signature.rsapss.sig.t.buffer));
414             if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
415                                                            &testDigest))
416                 SELF_TEST_FAILURE;
417         }
418         return result;
419     }
```

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*

```
420     static TPM_RC
421     TestRsa(
422         TPM_ALG_ID               alg,
423         ALGORITHM_VECTOR        *toTest
424         )
425     {
426         TPM_RC                   result = TPM_RC_SUCCESS;
427     //
428         switch(alg)
429         {
430             case ALG_NULL_VALUE:
431             // This is the RSAEP/RSADP function. If we are processing a list, don't
432             // need to test these now because any other test will validate
433             // RSAEP/RSADP. Can tell this is list of test by checking to see if
434             // 'toTest' is pointing at g_toTest. If so, this is an isolated test
435             // an need to go ahead and do the test;
436                 if((toTest == &g_toTest)
437                    || (!TEST_BIT(ALG_RSASSA_VALUE, *toTest)
438                        && !TEST_BIT(ALG_RSAES_VALUE, *toTest)
439                        && !TEST_BIT(ALG_RSAPSS_VALUE, *toTest)
440                        && !TEST_BIT(ALG_OAEP_VALUE, *toTest)))
441                    // Not running a list of tests or no other tests on the list
442                    // so run the test now
443                     result = TestRsaEncryptDecrypt(alg, toTest);
444                 // if not running the test now, leave the bit on, just in case things
445                 // get interrupted
446                 break;
447             case ALG_OAEP_VALUE:
448             case ALG_RSAES_VALUE:
449                 result = TestRsaEncryptDecrypt(alg, toTest);
450                 break;
451             case ALG_RSAPSS_VALUE:
452             case ALG_RSASSA_VALUE:
453                 result = TestRsaSignAndVerify(alg, toTest);
```

```
454             break;
455         default:
456             SELF_TEST_FAILURE;
457     }
458     return result;
459 }
460 #endif // ALG_RSA
```

10.2.1.6 ECC Tests

```
461 #if ALG_ECC
```

10.2.1.6.1 LoadEccParameter()

This function is mostly for readability and type checking

```
462 static void
463 LoadEccParameter(
464     TPM2B_ECC_PARAMETER          *to,       // target
465     const TPM2B_EC_TEST          *from      // source
466     )
467 {
468     MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
469 }
```

10.2.1.6.2 LoadEccPoint()

```
470 static void
471 LoadEccPoint(
472     TPMS_ECC_POINT               *point,    // target
473     const TPM2B_EC_TEST          *x,        // source
474     const TPM2B_EC_TEST          *y
475     )
476 {
477     MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
478     MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
479 }
```

10.2.1.6.3 TestECDH()

This test does a KVT on a point multiply.

```
480 static TPM_RC
481 TestECDH(
482     TPM_ALG_ID          scheme,         // IN: for consistency
483     ALGORITHM_VECTOR    *toTest         // IN/OUT: modified after test is run
484     )
485 {
486     static TPMS_ECC_POINT       Z;
487     static TPMS_ECC_POINT       Qe;
488     static TPM2B_ECC_PARAMETER  ds;
489     TPM_RC                      result = TPM_RC_SUCCESS;
490 //
491     NOT_REFERENCED(scheme);
492     CLEAR_BOTH(ALG_ECDH_VALUE);
493     LoadEccParameter(&ds, &c_ecTestKey_ds);
494     LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
495     if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds,
496                                                NULL, NULL))
497         SELF_TEST_FAILURE;
498     if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
```

```
499         || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
500             SELF_TEST_FAILURE;
501     return result;
502 }
```

10.2.1.6.4 TestEccSignAndVerify()

```
503 static TPM_RC
504 TestEccSignAndVerify(
505     TPM_ALG_ID                   scheme,
506     ALGORITHM_VECTOR            *toTest
507     )
508 {
509     static OBJECT                testObject;
510     static TPMT_SIGNATURE        testSig;
511     static TPMT_ECC_SCHEME       eccScheme;
512 
513     testSig.sigAlg = scheme;
514     testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
515 
516     eccScheme.scheme = scheme;
517     eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
518 
519     CLEAR_BOTH(scheme);
520     CLEAR_BOTH(ALG_ECDH_VALUE);
521 
522     // ECC signature verification testing uses a KVT.
523     switch(scheme)
524     {
525         case ALG_ECDSA_VALUE:
526             LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
527             LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
528             break;
529         case ALG_ECSCHNORR_VALUE:
530             LoadEccParameter(&testSig.signature.ecschnorr.signatureR,
531                              &c_TestEcSchnorr_r);
532             LoadEccParameter(&testSig.signature.ecschnorr.signatureS,
533                              &c_TestEcSchnorr_s);
534             break;
535         case ALG_SM2_VALUE:
536             // don't have a test for SM2
537             return TPM_RC_SUCCESS;
538         default:
539             SELF_TEST_FAILURE;
540             break;
541     }
542     TEST_DEFAULT_TEST_HASH(toTest);
543 
544     // Have to copy the key. This is because the size used in the test vectors
545     // is the size of the ECC parameter for the test key while the size of a point
546     // is TPM dependent
547     MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b,
548                  sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
549     LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX,
550                  &c_ecTestKey_QsY);
551     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
552 
553     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
554                                                    (TPM2B_DIGEST *)&c_ecTestValue.b))
555     {
556         SELF_TEST_FAILURE;
557     }
558     CHECK_CANCELED;
559 
560     // Now sign and verify some data
```

```
561         if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject,
562                                           (TPM2B_DIGEST *)&c_ecTestValue,
563                                           &eccScheme, NULL))
564             SELF_TEST_FAILURE;
565 
566         CHECK_CANCELED;
567 
568         if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
569                                                        (TPM2B_DIGEST *)&c_ecTestValue))
570             SELF_TEST_FAILURE;
571 
572         CHECK_CANCELED;
573 
574         return TPM_RC_SUCCESS;
575     }
```

10.2.1.6.5 TestKDFa()

```
576     static TPM_RC
577     TestKDFa(
578         ALGORITHM_VECTOR        *toTest
579         )
580     {
581         static TPM2B_KDF_TEST_KEY   keyOut;
582         UINT32                      counter = 0;
583     //
584         CLEAR_BOTH(ALG_KDF1_SP800_108_VALUE);
585 
586         keyOut.t.size = CryptKDFa(KDF_TEST_ALG, &c_kdfTestKeyIn.b, &c_kdfTestLabel.b,
587                                   &c_kdfTestContextU.b, &c_kdfTestContextV.b,
588                                   TEST_KDF_KEY_SIZE * 8, keyOut.t.buffer,
589                                   &counter, FALSE);
590         if (   keyOut.t.size != TEST_KDF_KEY_SIZE
591             || !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer,
592                             TEST_KDF_KEY_SIZE))
593             SELF_TEST_FAILURE;
594 
595         return TPM_RC_SUCCESS;
596     }
```

10.2.1.6.6 TestEcc()

```
597     static TPM_RC
598     TestEcc(
599         TPM_ALG_ID              alg,
600         ALGORITHM_VECTOR        *toTest
601         )
602     {
603         TPM_RC                  result = TPM_RC_SUCCESS;
604         NOT_REFERENCED(toTest);
605         switch(alg)
606         {
607             case ALG_ECC_VALUE:
608             case ALG_ECDH_VALUE:
609                 // If this is in a loop then see if another test is going to deal with
610                 // this.
611                 // If toTest is not a self-test list
612                 if((toTest == &g_toTest)
613                     // or this is the only ECC test in the list
614                    || !(TEST_BIT(ALG_ECDSA_VALUE, *toTest)
615                         || TEST_BIT(ALG_ECSCHNORR, *toTest)
616                         || TEST_BIT(ALG_SM2_VALUE, *toTest)))
617                 {
618                     result = TestECDH(alg, toTest);
```

```
619             }
620             break;
621         case ALG_ECDSA_VALUE:
622         case ALG_ECSCHNORR_VALUE:
623         case ALG_SM2_VALUE:
624             result = TestEccSignAndVerify(alg, toTest);
625             break;
626         default:
627             SELF_TEST_FAILURE;
628             break;
629     }
630     return result;
631 }
632 #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

```
633 LIB_EXPORT
634 TPM_RC
635 TestAlgorithm(
636     TPM_ALG_ID               alg,
637     ALGORITHM_VECTOR        *toTest
638     )
639 {
640     TPM_ALG_ID              first = (alg == ALG_ERROR_VALUE) ? ALG_FIRST_VALUE : alg;
641     TPM_ALG_ID              last = (alg == ALG_ERROR_VALUE) ? ALG_LAST_VALUE : alg;
642     BOOL                    doTest = (alg != ALG_ERROR_VALUE);
643     TPM_RC                  result = TPM_RC_SUCCESS;
644
645     if(toTest == NULL)
646         toTest = &g_toTest;
647
648     // This is kind of strange. This function will either run a test of the selected
649     // algorithm or just clear a bit if there is no test for the algorithm. So,
650     // either this loop will be executed once for the selected algorithm or once for
651     // each of the possible algorithms. If it is executed more than once ('alg' ==
652     // ALG_ERROR), then no test will be run but bits will be cleared for
653     // unimplemented algorithms. This was done this way so that there is only one
654     // case statement with all of the algorithms. It was easier to have one case
655     // statement than to have multiple ones to manage whenever an algorithm ID is
656     // added.
657     for(alg = first; (alg <= last); alg++)
658     {
659         // if 'alg' was TPM_ALG_ERROR, then we will be cycling through
660         // values, some of which may not be implemented. If the bit in toTest
661         // happens to be set, then we could either generated an assert, or just
662         // silently CLEAR it. Decided to just clear.
663         if(!TEST_BIT(alg, g_implementedAlgorithms))
```

```
664             {
665                 CLEAR_BIT(alg, *toTest);
666                 continue;
667             }
668             // Process whatever is left.
669             // NOTE: since this switch will only be called if the algorithm is
670             // implemented, it is not necessary to modify this list except to comment
671             // out the algorithms for which there is no test
672             switch(alg)
673             {
674             // Symmetric block ciphers
675     #if ALG_AES
676                 case ALG_AES_VALUE:
677     #endif  // ALG_AES
678     #if ALG_SM4
679                 // if SM4 is implemented, its test is like other block ciphers but there
680                 // aren't any test vectors for it yet
681     //            case ALG_SM4_VALUE:
682     #endif  // ALG_SM4
683     #if ALG_CAMELLIA
684                 // no test vectors for camellia
685     //            case ALG_CAMELLIA_VALUE:
686     #endif
687             // Symmetric modes
688     #if     !ALG_CFB
689     #   error   CFB is required in all TPM implementations
690     #endif // !ALG_CFB
691                 case ALG_CFB_VALUE:
692                     if(doTest)
693                         result = TestSymmetric(alg, toTest);
694                     break;
695     #if ALG_CTR
696                 case ALG_CTR_VALUE:
697     #endif // ALG_CRT
698     #if ALG_OFB
699                 case ALG_OFB_VALUE:
700     #endif // ALG_OFB
701     #if ALG_CBC
702                 case ALG_CBC_VALUE:
703     #endif // ALG_CBC
704     #if ALG_ECB
705                 case ALG_ECB_VALUE:
706     #endif
707                     if(doTest)
708                         result = TestSymmetric(alg, toTest);
709                     else
710                         // If doing the initialization of g_toTest vector, only need
711                         // to test one of the modes for the symmetric algorithms. If
712                         // initializing for a SelfTest(FULL_TEST), allow all the modes.
713                         if(toTest == &g_toTest)
714                             CLEAR_BIT(alg, *toTest);
715                     break;
716     #if     !ALG_HMAC
717     #   error   HMAC is required in all TPM implementations
718     #endif
719                 case ALG_HMAC_VALUE:
720                     // Clear the bit that indicates that HMAC is required because
721                     // HMAC is used as the basic test for all hash algorithms.
722                     CLEAR_BOTH(alg);
723                     // Testing HMAC means test the default hash
724                     if(doTest)
725                         TestHash(DEFAULT_TEST_HASH, toTest);
726                     else
727                         // If not testing, then indicate that the hash needs to be
728                         // tested because this uses HMAC
729                         SET_BOTH(DEFAULT_TEST_HASH);
```

```
730                 break;
731 #if ALG_SHA1
732             case ALG_SHA1_VALUE:
733 #endif // ALG_SHA1
734 #if ALG_SHA256
735             case ALG_SHA256_VALUE:
736 #endif // ALG_SHA256
737 #if ALG_SHA384
738             case ALG_SHA384_VALUE:
739 #endif // ALG_SHA384
740 #if ALG_SHA512
741             case ALG_SHA512_VALUE:
742 #endif // ALG_SHA512
743             // if SM3 is implemented its test is like any other hash, but there
744             // aren't any test vectors yet.
745 #if ALG_SM3_256
746 //            case ALG_SM3_256_VALUE:
747 #endif // ALG_SM3_256
748                 if(doTest)
749                     result = TestHash(alg, toTest);
750                 break;
751     // RSA-dependent
752 #if ALG_RSA
753             case ALG_RSA_VALUE:
754                 CLEAR_BOTH(alg);
755                 if(doTest)
756                     result = TestRsa(ALG_NULL_VALUE, toTest);
757                 else
758                     SET_BOTH(ALG_NULL_VALUE);
759                 break;
760             case ALG_RSASSA_VALUE:
761             case ALG_RSAES_VALUE:
762             case ALG_RSAPSS_VALUE:
763             case ALG_OAEP_VALUE:
764             case ALG_NULL_VALUE:    // used or RSADP
765                 if(doTest)
766                     result = TestRsa(alg, toTest);
767                 break;
768 #endif // ALG_RSA
769 #if ALG_KDF1_SP800_108
770             case ALG_KDF1_SP800_108_VALUE:
771                 if(doTest)
772                     result = TestKDFa(toTest);
773                 break;
774 #endif // ALG_KDF1_SP800_108
775 #if ALG_ECC
776     // ECC dependent but no tests
777     //        case ALG_ECDAA_VALUE:
778     //        case ALG_ECMQV_VALUE:
779     //        case ALG_KDF1_SP800_56a_VALUE:
780     //        case ALG_KDF2_VALUE:
781     //        case ALG_MGF1_VALUE:
782             case ALG_ECC_VALUE:
783                 CLEAR_BOTH(alg);
784                 if(doTest)
785                     result = TestEcc(ALG_ECDH_VALUE, toTest);
786                 else
787                     SET_BOTH(ALG_ECDH_VALUE);
788                 break;
789             case ALG_ECDSA_VALUE:
790             case ALG_ECDH_VALUE:
791             case ALG_ECSCHNORR_VALUE:
792 //            case ALG_SM2_VALUE:
793                 if(doTest)
794                     result = TestEcc(alg, toTest);
795                 break;
```

```
796     #endif // ALG_ECC
797                 default:
798                     CLEAR_BIT(alg, *toTest);
799                     break;
800             }
801             if(result != TPM_RC_SUCCESS)
802                 break;
803         }
804         return result;
805     }
806     #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
    }
```

```
    BnSetTop(bn, size);
    return bn;
}
```

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

```
LIB_EXPORT bigNum
BnFrom2B(
    bigNum           bn,          // OUT:
    const TPM2B     *a2B          // IN: number to convert
    )
{
    if(a2B != NULL)
        return BnFromBytes(bn, a2B->buffer, a2B->size);
    // Make sure that the number has an initialized value rather than whatever
    // was there before
    BnSetTop(bn, 0);    // Function accepts NULL
    return NULL;
}
```

10.2.2.3.3 BnFromHex()

Convert a hex string into a *bigNum*. This is primarily used in debugging.

```
LIB_EXPORT bigNum
BnFromHex(
    bigNum          bn,          // OUT:
    const char      *hex         // IN:
    )
{
#define FromHex(a)  ((a) - (((a) > 'a') ? ('a' + 10)                                \
                                        : ((a) > 'A') ? ('A' - 10) : '0'))
    unsigned             i;
    unsigned             wordCount;
    const char          *p;
    BYTE                *d = (BYTE *)&(bn->d[0]);
//
    pAssert(bn && hex);
    i = (unsigned)strlen(hex);
    wordCount = BYTES_TO_CRYPT_WORDS((i + 1) / 2);
    if((i == 0) || (wordCount >= BnGetAllocated(bn)))
        BnSetWord(bn, 0);
    else
    {
        bn->d[wordCount - 1] = 0;
        p = hex + i - 1;
        for(;i > 1; i -= 2)
        {
            BYTE a;
            a = FromHex(*p);
            p--;
            *d++ = a + (FromHex(*p) << 4);
            p--;
        }
        if(i == 1)
            *d = FromHex(*p);
    }
#if !BIG_ENDIAN_TPM
    for(i = 0; i < wordCount; i++)
        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}};
3
4   //** Functions
5
6   //*** AddSame()
7   // Adds two values that are the same size. This function allows 'result' to be
8   // the same as either of the addends. This is a nice function to put into assembly
9   // because handling the carry for multi-precision stuff is not as easy in C
10  // (unless there is a REALLY smart compiler). It would be nice if there were idioms
11  // in a language that a compiler could recognize what is going on and optimize
12  // loops like this.
13  //  Return Type: int
14  //      0           no carry out
15  //      1           carry out
16  static BOOL
17  AddSame(
18      crypt_uword_t           *result,
19      const crypt_uword_t     *op1,
20      const crypt_uword_t     *op2,
21      int                      count
22      )
23  {
24      int         carry = 0;
25      int         i;
26
27      for(i = 0; i < count; i++)
28      {
29          crypt_uword_t        a = op1[i];
30          crypt_uword_t        sum = a + op2[i];
31          result[i] = sum + carry;
```

```
32          // generate a carry if the sum is less than either of the inputs
33          // propagate a carry if there was a carry and the sum + carry is zero
34          // do this using bit operations rather than logical operations so that
35          // the time is about the same.
36          //              propagate term      | generate term
37          carry = ((result[i] == 0) & carry) | (sum < a);
38      }
39      return carry;
40  }
```

10.2.3.2.1 CarryProp()

Propagate a carry

```
41  static int
42  CarryProp(
43      crypt_uword_t           *result,
44      const crypt_uword_t     *op,
45      int                      count,
46      int                      carry
47      )
48  {
49      for(; count; count--)
50          carry = ((*result++ = *op++ + carry) == 0) & carry;
51      return carry;
52  }
53  static void
54  CarryResolve(
55      bigNum          result,
56      int             stop,
57      int             carry
58      )
59  {
60      if(carry)
61      {
62          pAssert((unsigned)stop < result->allocated);
63          result->d[stop++] = 1;
64      }
65      BnSetTop(result, stop);
66  }
```

10.2.3.2.2 BnAdd()

This function adds two *bigNum* values. This function always returns TRUE.

```
67  LIB_EXPORT BOOL
68  BnAdd(
69      bigNum           result,
70      bigConst         op1,
71      bigConst         op2
72      )
73  {
74      crypt_uword_t    stop;
75      int              carry;
76      const bignum_t   *n1 = op1;
77      const bignum_t   *n2 = op2;
78
79  //
80      if(n2->size > n1->size)
81      {
82          n1 = op2;
83          n2 = op1;
84      }
```

```
85      pAssert(result->allocated >= n1->size);
86      stop = MIN(n1->size, n2->allocated);
87      carry = (int)AddSame(result->d, n1->d, n2->d, (int)stop);
88      if(n1->size > stop)
89          carry = CarryProp(&result->d[stop], &n1->d[stop], (int)(n1->size - stop),
    carry);
90      CarryResolve(result, (int)n1->size, carry);
91      return TRUE;
92  }
```

10.2.3.2.3 BnAddWord()

This function adds a word value to a *bigNum*. This function always returns TRUE.

```
93  LIB_EXPORT BOOL
94  BnAddWord(
95      bigNum           result,
96      bigConst         op,
97      crypt_uword_t    word
98      )
99  {
100     int              carry;
101 //
102     carry = (result->d[0] = op->d[0] + word) < word;
103     carry = CarryProp(&result->d[1], &op->d[1], (int)(op->size - 1), carry);
104     CarryResolve(result, (int)op->size, carry);
105     return TRUE;
106 }
```

10.2.3.2.4 SubSame()

This function subtracts two values that have the same size.

```
107 static int
108 SubSame(
109     crypt_uword_t           *result,
110     const crypt_uword_t     *op1,
111     const crypt_uword_t     *op2,
112     int                      count
113     )
114 {
115     int                  borrow = 0;
116     int                  i;
117     for(i = 0; i < count; i++)
118     {
119         crypt_uword_t    a = op1[i];
120         crypt_uword_t    diff = a - op2[i];
121         result[i] = diff - borrow;
122         //       generate   |      propagate
123         borrow = (diff > a) | ((diff == 0) & borrow);
124     }
125     return borrow;
126 }
```

10.2.3.2.5 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.

```
127 static int
128 BorrowProp(
129     crypt_uword_t           *result,
130     const crypt_uword_t     *op,
131     int                      size,
132     int                      borrow
133     )
134 {
135     for(; size > 0; size--)
136         borrow = ((*result++ = *op++ - borrow) == MAX_CRYPT_UWORD) && borrow;
137     return borrow;
138 }
```

10.2.3.2.6 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.

```
139 LIB_EXPORT BOOL
140 BnSub(
141     bigNum           result,
142     bigConst         op1,
143     bigConst         op2
144     )
145 {
146     int             borrow;
147     int             stop = (int)MIN(op1->size, op2->allocated);
148 //
149     // Make sure that op2 is not obviously larger than op1
150     pAssert(op1->size >= op2->size);
151     borrow = SubSame(result->d, op1->d, op2->d, stop);
152     if(op1->size > (crypt_uword_t)stop)
153         borrow = BorrowProp(&result->d[stop], &op1->d[stop], (int)(op1->size - stop),
154                             borrow);
155     pAssert(!borrow);
156     BnSetTop(result, op1->size);
157     return TRUE;
158 }
```

10.2.3.2.7 BnSubWord()

This function subtracts a word value from a *bigNum*. This function always returns TRUE.

```
159 LIB_EXPORT BOOL
160 BnSubWord(
161     bigNum           result,
162     bigConst         op,
163     crypt_uword_t    word
164     )
165 {
166     int             borrow;
167 //
168     pAssert(op->size > 1 || word <= op->d[0]);
169     borrow = word > op->d[0];
170     result->d[0] = op->d[0] - word;
171     borrow = BorrowProp(&result->d[1], &op->d[1], (int)(op->size - 1), borrow);
172     pAssert(!borrow);
173     BnSetTop(result, op->size);
174     return TRUE;
175 }
```

10.2.3.2.8 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

```
176 LIB_EXPORT int
177 BnUnsignedCmp(
178     bigConst                op1,
179     bigConst                op2
180     )
181 {
182     int             retVal;
183     int             diff;
184     int             i;
185 //
186     pAssert((op1 != NULL) && (op2 != NULL));
187     retVal = (int)(op1->size - op2->size);
188     if(retVal == 0)
189     {
190         for(i = (int)(op1->size - 1); i >= 0; i--)
191         {
192             diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
193             retVal = retVal == 0 ? diff : retVal;
194         }
195     }
196     else
197         retVal = (retVal < 0) ? -1 : 1;
198     return retVal;
199 }
```

10.2.3.2.9 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

```
200 LIB_EXPORT int
201 BnUnsignedCmpWord(
202     bigConst            op1,
203     crypt_uword_t       word
204     )
205 {
206     if(op1->size > 1)
207         return 1;
208     else if(op1->size == 1)
209         return (op1->d[0] < word) ? -1 : (op1->d[0] > word);
210     else // op1 is zero
211         // equal if word is zero
212         return (word == 0) ? 0 : -1;
213 }
```

10.2.3.2.10 BnModWord()

This function does modular division of a big number when the modulus is a word value.

```
214   LIB_EXPORT crypt_word_t
215   BnModWord(
216       bigConst          numerator,
217       crypt_word_t      modulus
218       )
219   {
220       BN_MAX(remainder);
221       BN_VAR(mod, RADIX_BITS);
222   //
223       mod->d[0] = modulus;
224       mod->size = (modulus != 0);
225       BnDiv(NULL, remainder, numerator, mod);
226       return remainder->d[0];
227   }
```

10.2.3.2.11 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

```
228   LIB_EXPORT int
229   Msb(
230       crypt_uword_t           word
231       )
232   {
233       int             retVal = -1;
234   //
235   #if RADIX_BITS == 64
236       if(word & 0xffffffff00000000) { retVal += 32; word >>= 32; }
237   #endif
238       if(word & 0xffff0000) { retVal += 16; word >>= 16; }
239       if(word & 0x0000ff00) { retVal += 8; word >>= 8; }
240       if(word & 0x000000f0) { retVal += 4; word >>= 4; }
241       if(word & 0x0000000c) { retVal += 2; word >>= 2; }
242       if(word & 0x00000002) { retVal += 1; word >>= 1; }
243       return retVal + (int)word;
244   }
```

10.2.3.2.12 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

```
245   LIB_EXPORT int
246   BnMsb(
247       bigConst            bn
248       )
```

```
249 {
250     // If the value is NULL, or the size is zero then treat as zero and return -1
251     if(bn != NULL && bn->size > 0)
252     {
253         int         retVal = Msb(bn->d[bn->size - 1]);
254         retVal += (int)(bn->size - 1) * RADIX_BITS;
255         return retVal;
256     }
257     else
258         return -1;
259 }
```

10.2.3.2.13 BnSizeInBits()

This function returns the number of bits required to hold a number. It is one greater than the Msb.

```
260 LIB_EXPORT unsigned
261 BnSizeInBits(
262     bigConst                n
263     )
264 {
265     int     bits = BnMsb(n) + 1;
266 //
267     return bits < 0? 0 : (unsigned)bits;
268 }
```

10.2.3.2.14 BnSetWord()

Change the value of a bignum_t to a word value.

```
269 LIB_EXPORT bigNum
270 BnSetWord(
271     bigNum              n,
272     crypt_uword_t       w
273     )
274 {
275     if(n != NULL)
276     {
277         pAssert(n->allocated > 1);
278         n->d[0] = w;
279         BnSetTop(n, (w != 0) ? 1 : 0);
280     }
281     return n;
282 }
```

10.2.3.2.15 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

```
283 LIB_EXPORT BOOL
284 BnSetBit(
285     bigNum          bn,         // IN/OUT: big number to modify
286     unsigned int    bitNum      // IN: Bit number to SET
287     )
288 {
289     crypt_uword_t           offset = bitNum / RADIX_BITS;
290     pAssert(bn->allocated * RADIX_BITS >= bitNum);
291     // Grow the number if necessary to set the bit.
292     while(bn->size <= offset)
293         bn->d[bn->size++] = 0;
```

```
294         bn->d[offset] |= ((crypt_uword_t)1 << RADIX_MOD(bitNum));
295         return TRUE;
296     }
```

10.2.3.2.16 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

```
297     LIB_EXPORT BOOL
298     BnTestBit(
299         bigNum               bn,         // IN: number to check
300         unsigned int         bitNum      // IN: bit to test
301         )
302     {
303         crypt_uword_t         offset = RADIX_DIV(bitNum);
304     //
305         if(bn->size > offset)
306             return ((bn->d[offset] & (((crypt_uword_t)1) << RADIX_MOD(bitNum))) != 0);
307         else
308             return FALSE;
309     }
```

10.2.3.2.17 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

```
310     LIB_EXPORT BOOL
311     BnMaskBits(
312         bigNum           bn,         // IN/OUT: number to mask
313         crypt_uword_t    maskBit     // IN: the bit number for the mask.
314         )
315     {
316         crypt_uword_t    finalSize;
317         BOOL             retVal;
318
319         finalSize = BITS_TO_CRYPT_WORDS(maskBit);
320         retVal = (finalSize <= bn->allocated);
321         if(retVal && (finalSize > 0))
322         {
323             crypt_uword_t   mask;
324             mask = ~((crypt_uword_t)0) >> RADIX_MOD(maskBit);
325             bn->d[finalSize - 1] &= mask;
326         }
327         BnSetTop(bn, finalSize);
328         return retVal;
329     }
```

10.2.3.2.18 BnShiftRight()

This function will shift a *bigNum* to the right by the *shiftAmount*. This function always returns TRUE.

```
330  LIB_EXPORT BOOL
331  BnShiftRight(
332      bigNum           result,
333      bigConst         toShift,
334      uint32_t         shiftAmount
335      )
336  {
337      uint32_t         offset = (shiftAmount >> RADIX_LOG2);
338      uint32_t         i;
339      uint32_t         shiftIn;
340      crypt_uword_t    finalSize;
341  //
342      shiftAmount = shiftAmount & RADIX_MASK;
343      shiftIn = RADIX_BITS - shiftAmount;
344
345      // The end size is toShift->size - offset less one additional
346      // word if the shiftAmount would make the upper word == 0
347      if(toShift->size > offset)
348      {
349          finalSize = toShift->size - offset;
350          finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
351      }
352      else
353          finalSize = 0;
354
355      pAssert(finalSize <= result->allocated);
356      if(finalSize != 0)
357      {
358          for(i = 0; i < finalSize; i++)
359          {
360              result->d[i] = (toShift->d[i + offset] >> shiftAmount)
361                  | (toShift->d[i + offset + 1] << shiftIn);
362          }
363          if(offset == 0)
364              result->d[i] = toShift->d[i] >> shiftAmount;
365      }
366      BnSetTop(result, finalSize);
367      return TRUE;
368  }
```

10.2.3.2.19 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

```
369  LIB_EXPORT BOOL
370  BnGetRandomBits(
371      bigNum           n,
372      size_t           bits,
```

```
373         RAND_STATE      *rand
374     )
375     {
376         // Since this could be used for ECC key generation using the extra bits method,
377         // make sure that the value is large enough
378         TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
379         TPM2B_LARGEST    large;
380     //
381         large.b.size = (UINT16)BITS_TO_BYTES(bits);
382         if(DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size)
383         {
384             if(BnFrom2B(n, &large.b) != NULL)
385             {
386                 if(BnMaskBits(n, (crypt_uword_t)bits))
387                     return TRUE;
388             }
389         }
390         return FALSE;
391     }
```

10.2.3.2.20 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)

```
392     LIB_EXPORT BOOL
393     BnGenerateRandomInRange(
394         bigNum           dest,
395         bigConst         limit,
396         RAND_STATE      *rand
397         )
398     {
399         size_t   bits = BnSizeInBits(limit);
400     //
401         if(bits < 2)
402         {
403             BnSetWord(dest, 0);
404             return FALSE;
405         }
406         else
407         {
408             while(BnGetRandomBits(dest, bits, rand)
409                   && (BnEqualZero(dest) || (BnUnsignedCmp(dest, limit) >= 0)));
410         }
411         return !g_inFailureMode;
412     }
```

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

```
#define _CRYPT_HASH_C_
#include "Tpm.h"
#include "CryptSym.h"
#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.

```
UINT16
CryptCmacStart(
    SMAC_STATE          *state,
    TPMU_PUBLIC_PARMS   *keyParms,
    TPM_ALG_ID           macAlg,
    TPM2B               *key
)
{
    tpmCmacState_t      *cState = &state->state.cmac;
    TPMT_SYM_DEF_OBJECT *def = &keyParms->symDetail.sym;
//
    if(macAlg != TPM_ALG_CMAC)
        return 0;
    // set up the encryption algorithm and parameters
    cState->symAlg = def->algorithm;
    cState->keySizeBits = def->keyBits.sym;
    cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm,
                                                   def->keyBits.sym);
    MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));

    // Set up the dispatch methods for the CMAC
    state->smacMethods.data = CryptCmacData;
    state->smacMethods.end = CryptCmacEnd;
    return cState->iv.t.size;
}
```

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                   digestSize;
68
69      scheme = &publicArea->parameters.keyedHashDetail.scheme;
70
71      if(publicArea->type != ALG_KEYEDHASH_VALUE)
72          return TPM_RC_FAILURE;
73
74      // Pick the limiting hash algorithm
75      if(scheme->scheme == ALG_NULL_VALUE)
76          hashAlg = publicArea->nameAlg;
77      else if(scheme->scheme == ALG_XOR_VALUE)
78          hashAlg = scheme->details.xor.hashAlg;
```

```
79          else
80              hashAlg = scheme->details.hmac.hashAlg;
81          digestSize = CryptHashGetDigestSize(hashAlg);
82
83          // if this is a signing or a decryption key, then the limit
84          // for the data size is the block size of the hash. This limit
85          // is set because larger values have lower entropy because of the
86          // HMAC function. The lower limit is 1/2 the size of the digest
87          //
88          //If the user provided the key, check that it is a proper size
89          if(sensitiveCreate->data.t.size != 0)
90          {
91              if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
92                 || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
93              {
94                  if(sensitiveCreate->data.t.size > CryptHashGetBlockSize(hashAlg))
95                      return TPM_RC_SIZE;
96  #if 0       // May make this a FIPS-mode requirement
97                  if(sensitiveCreate->data.t.size < (digestSize / 2))
98                      return TPM_RC_SIZE;
99  #endif
100             }
101             // If this is a data blob, then anything that will get past the unmarshaling
102             // is OK
103             MemoryCopy2B(&sensitive->sensitive.bits.b, &sensitiveCreate->data.b,
104                          sizeof(sensitive->sensitive.bits.t.buffer));
105         }
106         else
107         {
108             // The TPM is going to generate the data so set the size to be the
109             // size of the digest of the algorithm
110             sensitive->sensitive.bits.t.size =
111                 DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
112             if(sensitive->sensitive.bits.t.size == 0)
113                 return (g_inFailureMode) ? TPM_RC_FAILURE : TPM_RC_NO_RESULT;
114         }
115         return TPM_RC_SUCCESS;
116 }
```

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.

```
117 BOOL
118 CryptIsSchemeAnonymous(
119     TPM_ALG_ID       scheme          // IN: the scheme algorithm to test
120     )
121 {
122     return scheme == ALG_ECDAA_VALUE;
123 }
```

10.2.6.4 Symmetric Functions

10.2.6.4.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
124 void
125 ParmDecryptSym(
126     TPM_ALG_ID       symAlg,         // IN: the symmetric algorithm
127     TPM_ALG_ID       hash,           // IN: hash algorithm for KDFa
```

```
128         UINT16           keySizeInBits, // IN: the key size in bits
129         TPM2B           *key,           // IN: KDF HMAC key
130         TPM2B           *nonceCaller,   // IN: nonce caller
131         TPM2B           *nonceTpm,      // IN: nonce TPM
132         UINT32           dataSize,      // IN: size of parameter buffer
133         BYTE            *data           // OUT: buffer to be decrypted
134         )
135     {
136         // KDF output buffer
137         // It contains parameters for the CFB encryption
138         // From MSB to LSB, they are the key and iv
139         BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
140         // Symmetric key size in byte
141         UINT16           keySize = (keySizeInBits + 7) / 8;
142         TPM2B_IV         iv;
143
144         iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
145         // If there is decryption to do...
146         if(iv.t.size > 0)
147         {
148             // Generate key and iv
149             CryptKDFa(hash, key, CFB_KEY, nonceCaller, nonceTpm,
150                       keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
151             MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
152
153             CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString,
154                                   &iv, ALG_CFB_VALUE, dataSize, data);
155         }
156         return;
157     }
```

10.2.6.4.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
158     void
159     ParmEncryptSym(
160         TPM_ALG_ID       symAlg,        // IN: symmetric algorithm
161         TPM_ALG_ID       hash,          // IN: hash algorithm for KDFa
162         UINT16           keySizeInBits, // IN: symmetric key size in bits
163         TPM2B           *key,           // IN: KDF HMAC key
164         TPM2B           *nonceCaller,   // IN: nonce caller
165         TPM2B           *nonceTpm,      // IN: nonce TPM
166         UINT32           dataSize,      // IN: size of parameter buffer
167         BYTE            *data           // OUT: buffer to be encrypted
168         )
169     {
170         // KDF output buffer
171         // It contains parameters for the CFB encryption
172         BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
173
174         // Symmetric key size in bytes
175         UINT16           keySize = (keySizeInBits + 7) / 8;
176
177         TPM2B_IV         iv;
178
179         iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
180         // See if there is any encryption to do
181         if(iv.t.size > 0)
182         {
183             // Generate key and iv
184             CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller,
185                       keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
186             MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
```

```
187
188             CryptSymmetricEncrypt(data, symAlg, keySizeInBits, symParmString, &iv,
189                                   ALG_CFB_VALUE, dataSize, data);
190         }
191         return;
192     }
```

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

```
193     static TPM_RC
194     CryptGenerateKeySymmetric(
195         TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
196                                                     //     for the new key.
197         TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
198         TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
199         RAND_STATE              *rand               // IN: the "entropy" source for
200         )
201     {
202         UINT16           keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
203         TPM_RC           result;
204     //
205         // only do multiples of RADIX_BITS
206         if((keyBits % RADIX_BITS) != 0)
207             return TPM_RC_KEY_SIZE;
208         // If this is not a new key, then the provided key data must be the right size
209         if(sensitiveCreate->data.t.size != 0)
210         {
211             result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
212                                          (TPM2B_SYM_KEY *)&sensitiveCreate->data);
213             if(result == TPM_RC_SUCCESS)
214                 MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
215                              sizeof(sensitive->sensitive.sym.t.buffer));
216         }
217     #if ALG_TDES
218         else if(publicArea->parameters.symDetail.sym.algorithm == ALG_TDES_VALUE)
219         {
220             result = CryptGenerateKeyDes(publicArea, sensitive, rand);
221         }
222     #endif
223         else
224         {
225             sensitive->sensitive.sym.t.size =
226                 DRBG_Generate(rand, sensitive->sensitive.sym.t.buffer,
227                               BITS_TO_BYTES(keyBits));
228             if(g_inFailureMode)
229                 result = TPM_RC_FAILURE;
230             else if(sensitive->sensitive.sym.t.size == 0)
231                 result = TPM_RC_NO_RESULT;
232             else
233                 result = TPM_RC_SUCCESS;
234         }
235         return result;
236     }
```

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.

```
void
CryptXORObfuscation(
    TPM_ALG_ID       hash,          // IN: hash algorithm for KDF
    TPM2B           *key,           // IN: KDF key
    TPM2B           *contextU,      // IN: contextU
    TPM2B           *contextV,      // IN: contextV
    UINT32           dataSize,      // IN: size of data buffer
    BYTE            *data           // IN/OUT: data to be XORed in place
    )
{
    BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
    BYTE            *pm;
    UINT32           i;
    UINT32           counter = 0;
    UINT16           hLen = CryptHashGetDigestSize(hash);
    UINT32           requestSize = dataSize * 8;
    INT32            remainBytes = (INT32)dataSize;

    pAssert((key != NULL) && (data != NULL) && (hLen != 0));

    // Call KDFa to generate XOR mask
    for(; remainBytes > 0; remainBytes -= hLen)
    {
        // Make a call to KDFa to get next iteration
        CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
                  requestSize, mask, &counter, TRUE);

    // XOR next piece of the data
        pm = mask;
        for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
            *data++ ^= *pm++;
    }
    return;
}
```

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

```
BOOL
CryptInit(
    void
    )
{
    BOOL         ok;
    // Initialize the vector of implemented algorithms
```

```
278         AlgorithmGetImplementedVector(&g_implementedAlgorithms);
279
280         // Indicate that all test are necessary
281         CryptInitializeToTest();
282
283         // Do any library initializations that are necessary. If any fails,
284         // the caller should go into failure mode;
285         ok = SupportLibInit();
286         ok = ok && CryptSymInit();
287         ok = ok && CryptRandInit();
288         ok = ok && CryptHashInit();
289     #if ALG_RSA
290         ok = ok && CryptRsaInit();
291     #endif // ALG_RSA
292     #if ALG_ECC
293         ok = ok && CryptEccInit();
294     #endif // ALG_ECC
295         return ok;
296     }
```

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

```
297     BOOL
298     CryptStartup(
299         STARTUP_TYPE     type           // IN: the startup type
300         )
301     {
302         BOOL             OK;
303         NOT_REFERENCED(type);
304
305         OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
306     #if ALG_RSA
307             &&  CryptRsaStartup()
308     #endif // ALG_RSA
309     #if ALG_ECC
310             &&  CryptEccStartup()
311     #endif // ALG_ECC
312             ;
313     #if ALG_ECC
314         // Don't directly check for SU_RESET because that is the default
315         if(OK && (type != SU_RESTART) && (type != SU_RESUME))
316         {
317         // If the shutdown was orderly, then the values recovered from NV will
318         // be OK to use.
319             // Get a new  random commit nonce
320             gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
321             CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
322             // Reset the counter and commit array
323             gr.commitCounter = 0;
324             MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
325         }
326     #endif // ALG_ECC
327         return OK;
328     }
```

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

```
329  BOOL
330  CryptIsAsymAlgorithm(
331      TPM_ALG_ID        algID           // IN: algorithm ID
332      )
333  {
334      switch(algID)
335      {
336  #if ALG_RSA
337          case ALG_RSA_VALUE:
338  #endif
339  #if ALG_ECC
340          case ALG_ECC_VALUE:
341  #endif
342              return TRUE;
343              break;
344          default:
345              break;
346      }
347      return FALSE;
348  }
```

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

```
349  TPM_RC
350  CryptSecretEncrypt(
351      OBJECT                  *encryptKey,    // IN: encryption key object
352      const TPM2B             *label,         // IN: a null-terminated string as L
353      TPM2B_DATA              *data,          // OUT: secret value
354      TPM2B_ENCRYPTED_SECRET  *secret         // OUT: secret structure
355      )
```

```
356 {
357     TPMT_RSA_DECRYPT         scheme;
358     TPM_RC                   result = TPM_RC_SUCCESS;
359 //
360     if(data == NULL || secret == NULL)
361         return TPM_RC_FAILURE;
362
363     // The output secret value has the size of the digest produced by the nameAlg.
364     data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
365     // The encryption scheme is OAEP using the nameAlg of the encrypt key.
366     scheme.scheme = ALG_OAEP_VALUE;
367     scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
368
369     if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
370         return TPM_RC_ATTRIBUTES;
371     switch(encryptKey->publicArea.type)
372     {
373 #if ALG_RSA
374         case ALG_RSA_VALUE:
375         {
376             // Create secret data from RNG
377             CryptRandomGenerate(data->t.size, data->t.buffer);
378
379             // Encrypt the data by RSA OAEP into encrypted secret
380             result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
381                                      encryptKey, &scheme, label, NULL);
382         }
383         break;
384 #endif // ALG_RSA
385
386 #if ALG_ECC
387         case ALG_ECC_VALUE:
388         {
389             TPMS_ECC_POINT      eccPublic;
390             TPM2B_ECC_PARAMETER eccPrivate;
391             TPMS_ECC_POINT      eccSecret;
392             BYTE                *buffer = secret->t.secret;
393
394             // Need to make sure that the public point of the key is on the
395             // curve defined by the key.
396             if(!CryptEccIsPointOnCurve(
397                 encryptKey->publicArea.parameters.eccDetail.curveID,
398                 &encryptKey->publicArea.unique.ecc))
399                 result = TPM_RC_KEY;
400             else
401             {
402                 // Call crypto engine to create an auxiliary ECC key
403                 // We assume crypt engine initialization should always success.
404                 // Otherwise, TPM should go to failure mode.
405
406                 CryptEccNewKeyPair(&eccPublic, &eccPrivate,
407                                    encryptKey->publicArea.parameters.eccDetail.curveID);
408                 // Marshal ECC public to secret structure. This will be used by the
409                 // recipient to decrypt the secret with their private key.
410                 secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
411
412                 // Compute ECDH shared secret which is R = [d]Q where d is the
413                 // private part of the ephemeral key and Q is the public part of a
414                 // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
415                 // because the auxiliary ECC key is just created according to the
416                 // parameters of input ECC encrypt key.
417                 if(CryptEccPointMultiply(&eccSecret,
418                             encryptKey->publicArea.parameters.eccDetail.curveID,
419                             &encryptKey->publicArea.unique.ecc, &eccPrivate,
420                             NULL, NULL)
421                    != TPM_RC_SUCCESS)
```

```
422                     result = TPM_RC_KEY;
423                 else
424                 {
425                     // The secret value is computed from Z using KDFe as:
426                     // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
427                     // Where:
428                     //  HashID  the nameAlg of the decrypt key
429                     //  Z    the x coordinate (Px) of the product (P) of the point
430                     //       (Q) of the secret and the private x coordinate (de,V)
431                     //       of the decryption key
432                     //  Use a null-terminated string containing "SECRET"
433                     //  PartyUInfo  the x coordinate of the point in the secret
434                     //              (Qe,U )
435                     //  PartyVInfo  the x coordinate of the public key (Qs,V )
436                     //  bits    the number of bits in the digest of HashID
437                     // Retrieve seed from KDFe
438                     CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
439                               label, &eccPublic.x.b,
440                               &encryptKey->publicArea.unique.ecc.x.b,
441                               data->t.size * 8, data->t.buffer);
442                 }
443             }
444         }
445         break;
446 #endif // ALG_ECC
447         default:
448             FAIL(FATAL_ERROR_INTERNAL);
449             break;
450     }
451     return result;
452 }
```

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

```
453 TPM_RC
454 CryptSecretDecrypt(
455     OBJECT                  *decryptKey,    // IN: decrypt key
456     TPM2B_NONCE             *nonceCaller,   // IN: nonceCaller.  It is needed for
457                                             //     symmetric decryption.  For
458                                             //     asymmetric decryption, this
```

```
459                                             //     parameter is NULL
460     const TPM2B             *label,         // IN: a value for L
461     TPM2B_ENCRYPTED_SECRET  *secret,        // IN: input secret
462     TPM2B_DATA              *data           // OUT: decrypted secret value
463     )
464 {
465     TPM_RC      result = TPM_RC_SUCCESS;
466
467     // Decryption for secret
468     switch(decryptKey->publicArea.type)
469     {
470 #if ALG_RSA
471         case ALG_RSA_VALUE:
472         {
473             TPMT_RSA_DECRYPT        scheme;
474             TPMT_RSA_SCHEME         *keyScheme
475                 = &decryptKey->publicArea.parameters.rsaDetail.scheme;
476             UINT16                   digestSize;
477
478             scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
479             // If the key scheme is ALG_NULL_VALUE, set the scheme to OAEP and
480             // set the algorithm to the name algorithm.
481             if(scheme.scheme == ALG_NULL_VALUE)
482             {
483                 // Use OAEP scheme
484                 scheme.scheme = ALG_OAEP_VALUE;
485                 scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
486             }
487             // use the digestSize as an indicator of whether or not the scheme
488             // is using a supported hash algorithm.
489             // Note: depending on the scheme used for encryption, a hashAlg might
490             // not be needed. However, the return value has to have some upper
491             // limit on the size. In this case, it is the size of the digest of the
492             // hash algorithm. It is checked after the decryption is done but, there
493             // is no point in doing the decryption if the size is going to be
494             // 'wrong' anyway.
495             digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
496             if(scheme.scheme != ALG_OAEP_VALUE || digestSize == 0)
497                 return TPM_RC_SCHEME;
498
499             // Set the output buffer capacity
500             data->t.size = sizeof(data->t.buffer);
501
502             // Decrypt seed by RSA OAEP
503             result = CryptRsaDecrypt(&data->b, &secret->b,
504                                      decryptKey, &scheme, label);
505             if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
506                 result = TPM_RC_VALUE;
507         }
508         break;
509 #endif // ALG_RSA
510 #if ALG_ECC
511         case ALG_ECC_VALUE:
512         {
513             TPMS_ECC_POINT       eccPublic;
514             TPMS_ECC_POINT       eccSecret;
515             BYTE                *buffer = secret->t.secret;
516             INT32                size = secret->t.size;
517
518             // Retrieve ECC point from secret buffer
519             result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
520             if(result == TPM_RC_SUCCESS)
521             {
522                 result = CryptEccPointMultiply(&eccSecret,
523                             decryptKey->publicArea.parameters.eccDetail.curveID,
524                             &eccPublic, &decryptKey->sensitive.sensitive.ecc,
```

```
                                NULL, NULL);
                if(result == TPM_RC_SUCCESS)
                {
                    // Set the size of the "recovered" secret value to be the size
                    // of the digest produced by the nameAlg.
                    data->t.size =
                        CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);

                // The secret value is computed from Z using KDFe as:
                // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
                // Where:
                //  HashID -- the nameAlg of the decrypt key
                //  Z --  the x coordinate (Px) of the product (P) of the point
                //        (Q) of the secret and the private x coordinate (de,V)
                //        of the decryption key
                //  Use -- a null-terminated string containing "SECRET"
                //  PartyUInfo -- the x coordinate of the point in the secret
                //                (Qe,U )
                //  PartyVInfo -- the x coordinate of the public key (Qs,V )
                //  bits -- the number of bits in the digest of HashID
                // Retrieve seed from KDFe
                    CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
                              &eccPublic.x.b,
                              &decryptKey->publicArea.unique.ecc.x.b,
                              data->t.size * 8, data->t.buffer);
                }
            }
        }
        break;
#endif // ALG_ECC
#if     !ALG_KEYEDHASH
#   error   "KEYEDHASH support is required"
#endif
        case ALG_KEYEDHASH_VALUE:
            // The seed size can not be bigger than the digest size of nameAlg
            if(secret->t.size >
               CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
                result = TPM_RC_VALUE;
            else
            {
                // Retrieve seed by XOR Obfuscation:
                //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
                //    where:
                //    secret  the secret parameter from the TPM2_StartAuthHMAC
                //            command that contains the seed value
                //    hash    nameAlg  of tpmKey
                //    key     the key or data value in the object referenced by
                //            entityHandle in the TPM2_StartAuthHMAC command
                //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
                //    nullNonce   a zero-length nonce
                // XOR Obfuscation in place
                CryptXORObfuscation(decryptKey->publicArea.nameAlg,
                                    &decryptKey->sensitive.sensitive.bits.b,
                                    &nonceCaller->b, NULL,
                                    secret->t.size, secret->t.secret);
                // Copy decrypted seed
                MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
            }
            break;
        case ALG_SYMCIPHER_VALUE:
        {
            TPM2B_IV                iv = {{0}};
            TPMT_SYM_DEF_OBJECT     *symDef;
            // The seed size can not be bigger than the digest size of nameAlg
            if(secret->t.size >
               CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
```

```
591                 result = TPM_RC_VALUE;
592             else
593             {
594                 symDef = &decryptKey->publicArea.parameters.symDetail.sym;
595                 iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
596                                                        symDef->keyBits.sym);
597                 if(iv.t.size == 0)
598                     return TPM_RC_FAILURE;
599                 if(nonceCaller->t.size >= iv.t.size)
600                 {
601                     MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
602                 }
603                 else
604                 {
605                     if(nonceCaller->t.size > sizeof(iv.t.buffer))
606                         return TPM_RC_FAILURE;
607                     MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
608                                nonceCaller->t.size);
609                 }
610                 // make sure secret will fit
611                 if(secret->t.size > data->t.size)
612                     return TPM_RC_FAILURE;
613                 data->t.size = secret->t.size;
614                 // CFB decrypt, using nonceCaller as iv
615                 CryptSymmetricDecrypt(data->t.buffer, symDef->algorithm,
616                                       symDef->keyBits.sym,
617                                       decryptKey->sensitive.sensitive.sym.t.buffer,
618                                       &iv, ALG_CFB_VALUE, secret->t.size,
619                                       secret->t.secret);
620             }
621         }
622         break;
623         default:
624             FAIL(FATAL_ERROR_INTERNAL);
625             break;
626     }
627     return result;
628 }
```

10.2.6.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
629 void
630 CryptParameterEncryption(
631     TPM_HANDLE       handle,            // IN: encrypt session handle
632     TPM2B           *nonceCaller,       // IN: nonce caller
633     UINT16           leadingSizeInByte, // IN: the size of the leading size field in
634                                         //     bytes
635     TPM2B_AUTH      *extraKey,          // IN: additional key material other than
636                                         //     sessionAuth
637     BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
638     )
639 {
640     SESSION     *session = SessionGet(handle);  // encrypt session
641     TPM2B_TYPE(TEMP_KEY, (sizeof(extraKey->t.buffer)
642                           + sizeof(session->sessionKey.t.buffer)));
643     TPM2B_TEMP_KEY        key;              // encryption key
644     UINT32                cipherSize = 0;   // size of cipher text
645 //
646     // Retrieve encrypted data size.
647     if(leadingSizeInByte == 2)
648     {
649         // Extract the first two bytes as the size field as the data size
```

```
650             // encrypt
651             cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
652             // advance the buffer
653             buffer = &buffer[2];
654         }
655     #ifdef      TPM4B
656         else if(leadingSizeInByte == 4)
657         {
658             // use the first four bytes to indicate the number of bytes to encrypt
659             cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
660             //advance pointer
661             buffer = &buffer[4];
662         }
663     #endif
664         else
665         {
666             FAIL(FATAL_ERROR_INTERNAL);
667         }
668
669         // Compute encryption key by concatenating sessionKey with extra key
670         MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
671         MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
672
673         if(session->symmetric.algorithm == ALG_XOR_VALUE)
674
675             // XOR parameter encryption formulation:
676             //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
677             CryptXORObfuscation(session->authHashAlg, &(key.b),
678                                 &(session->nonceTPM.b),
679                                 nonceCaller, cipherSize, buffer);
680         else
681             ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
682                            session->symmetric.keyBits.aes, &(key.b),
683                            nonceCaller, &(session->nonceTPM.b),
684                            cipherSize, buffer);
685         return;
686     }
```

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.

```
687     TPM_RC
688     CryptParameterDecryption(
689         TPM_HANDLE       handle,            // IN: encrypted session handle
690         TPM2B           *nonceCaller,       // IN: nonce caller
691         UINT32           bufferSize,        // IN: size of parameter buffer
692         UINT16           leadingSizeInByte, // IN: the size of the leading size field in
693                                             //     byte
694         TPM2B_AUTH      *extraKey,          // IN: the authValue
695         BYTE            *buffer             // IN/OUT: parameter buffer to be decrypted
696         )
697     {
698         SESSION         *session = SessionGet(handle);  // encrypt session
699         // The HMAC key is going to be the concatenation of the session key and any
700         // additional key material (like the authValue). The size of both of these
701         // is the size of the buffer which can contain a TPMT_HA.
702         TPM2B_TYPE(HMAC_KEY, (sizeof(extraKey->t.buffer)
703                               + 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;
```

```
763             unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
764                                               &sensitive->seedValue.b);
765             CryptDigestUpdate2B(&hmacState.hashState,
766                                 &sensitive->sensitive.any.b);
767             CryptHmacEnd2B(&hmacState, &unique->b);
768         }
769         else
770         {
771             HASH_STATE  hashState;
772             // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
773             unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
774             CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
775             CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
776             CryptHashEnd2B(&hashState, &unique->b);
777         }
778         return;
779     }
```

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

```
780 TPM_RC
781 CryptCreateObject(
782     OBJECT                  *object,            // IN: new object structure pointer
783     TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation
784     RAND_STATE              *rand               // IN: the random number generator
785                                                 //      to use
786     )
```

```
787 {
788     TPMT_PUBLIC             *publicArea = &object->publicArea;
789     TPMT_SENSITIVE          *sensitive = &object->sensitive;
790     TPM_RC                   result = TPM_RC_SUCCESS;
791 //
792     // Set the sensitive type for the object
793     sensitive->sensitiveType = publicArea->type;
794
795     // For all objects, copy the initial authorization data
796     sensitive->authValue = sensitiveCreate->userAuth;
797
798     // If the TPM is the source of the data, set the size of the provided data to
799     // zero so that there's no confusion about what to do.
800     if(IS_ATTRIBUTE(publicArea->objectAttributes,
801                     TPMA_OBJECT, sensitiveDataOrigin))
802         sensitiveCreate->data.t.size = 0;
803
804     // Generate the key and unique fields for the asymmetric keys and just the
805     // sensitive value for symmetric object
806     switch(publicArea->type)
807     {
808 #if ALG_RSA
809         // Create RSA key
810         case ALG_RSA_VALUE:
811             // RSA uses full object so that it has a place to put the private
812             // exponent
813             result = CryptRsaGenerateKey(publicArea, sensitive, rand);
814             break;
815 #endif // ALG_RSA
816
817 #if ALG_ECC
818         // Create ECC key
819         case ALG_ECC_VALUE:
820             result = CryptEccGenerateKey(publicArea, sensitive, rand);
821             break;
822 #endif // ALG_ECC
823         case ALG_SYMCIPHER_VALUE:
824             result = CryptGenerateKeySymmetric(publicArea, sensitive,
825                                                sensitiveCreate, rand);
826             break;
827         case ALG_KEYEDHASH_VALUE:
828             result = CryptGenerateKeyedHash(publicArea, sensitive,
829                                             sensitiveCreate, rand);
830             break;
831         default:
832             FAIL(FATAL_ERROR_INTERNAL);
833             break;
834     }
835     if(result != TPM_RC_SUCCESS)
836         return result;
837 // Create the sensitive seed value
838     // If this is a primary key in the endorsement hierarchy, stir the DRBG state
839     // This implementation uses both shProof and ehProof to make sure that there
840     // is no leakage of either.
841     if(object->attributes.primary && object->attributes.epsHierarchy)
842     {
843         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
844         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.ehProof.b);
845     }
846     // Generate a seedValue that is the size of the digest produced by nameAlg
847     sensitive->seedValue.t.size =
848         DRBG_Generate(rand, sensitive->seedValue.t.buffer,
849                       CryptHashGetDigestSize(publicArea->nameAlg));
850     if(g_inFailureMode)
851         return TPM_RC_FAILURE;
852     else if(sensitive->seedValue.t.size == 0)
```

```
853             return TPM_RC_NO_RESULT;
854         // For symmetric objects, need to compute the unique value for the public area
855         if(publicArea->type == ALG_SYMCIPHER_VALUE
856            || publicArea->type == ALG_KEYEDHASH_VALUE)
857         {
858             CryptComputeSymmetricUnique(publicArea, sensitive, &publicArea->unique.sym);
859         }
860         else
861         {
862             // if this is an asymmetric key and it isn't a parent, then
863             // get rid of the seed.
864             if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
865                || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
866                 memset(&sensitive->seedValue, 0, sizeof(sensitive->seedValue));
867         }
868         // Compute the name
869         PublicMarshalAndComputeName(publicArea, &object->name);
870         return result;
871     }
```

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

```
872     TPMI_ALG_HASH
873     CryptGetSignHashAlg(
874         TPMT_SIGNATURE  *auth           // IN: signature
875         )
876     {
877         if(auth->sigAlg == ALG_NULL_VALUE)
878             FAIL(FATAL_ERROR_INTERNAL);
879
880         // Get authHash algorithm based on signing scheme
881         switch(auth->sigAlg)
882         {
883     #if ALG_RSA
884         // If RSA is supported, both RSASSA and RSAPSS are required
885     #   if !defined ALG_RSASSA_VALUE || !defined ALG_RSAPSS_VALUE
886     #       error "RSASSA and RSAPSS are required for RSA"
887     #   endif
888             case ALG_RSASSA_VALUE:
889                 return auth->signature.rsassa.hash;
890             case ALG_RSAPSS_VALUE:
891                 return auth->signature.rsapss.hash;
892     #endif // ALG_RSA
893
894     #if ALG_ECC
895         // If ECC is defined, ECDSA is mandatory
896     #   if !ALG_ECDSA
897     #       error "ECDSA is requried for ECC"
898     #   endif
899             case ALG_ECDSA_VALUE:
900         // SM2 and ECSCHNORR are optional
901
902     #   if      ALG_SM2
903             case ALG_SM2_VALUE:
904     #   endif
905     #   if  ALG_ECSCHNORR
906             case ALG_ECSCHNORR_VALUE:
907     #   endif
908             //all ECC signatures look the same
909                 return auth->signature.ecdsa.hash;
910
```

```
911     #   if  ALG_ECDAA
912             // Don't know how to verify an ECDAA signature
913             case ALG_ECDAA_VALUE:
914                 break;
915     #   endif
916
917     #endif // ALG_ECC
918
919             case ALG_HMAC_VALUE:
920                 return auth->signature.hmac.hashAlg;
921
922             default:
923                 break;
924         }
925         return ALG_NULL_VALUE;
926     }
```

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().

```
927     BOOL
928     CryptIsSplitSign(
929         TPM_ALG_ID       scheme          // IN: the algorithm selector
930         )
931     {
932         switch(scheme)
933         {
934     #   if  ALG_ECDAA
935             case ALG_ECDAA_VALUE:
936                 return TRUE;
937                 break;
938     #   endif   // ALG_ECDAA
939             default:
940                 return FALSE;
941                 break;
942         }
943     }
```

10.2.6.6.11 CryptIsAsymSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
944     BOOL
945     CryptIsAsymSignScheme(
946         TPMI_ALG_PUBLIC          publicType,        // IN: Type of the object
947         TPMI_ALG_ASYM_SCHEME     scheme             // IN: the scheme
948         )
949     {
950         BOOL            isSignScheme = TRUE;
951
952         switch(publicType)
953         {
954     #if ALG_RSA
955             case ALG_RSA_VALUE:
956                 switch(scheme)
957                 {
958     #   if !ALG_RSASSA  || !ALG_RSAPSS
959     #       error "RSASSA and PSAPSS required if RSA used."
960     #   endif
961                     case ALG_RSASSA_VALUE:
962                     case ALG_RSAPSS_VALUE:
```

```
963                     break;
964                 default:
965                     isSignScheme = FALSE;
966                     break;
967             }
968             break;
969 #endif // ALG_RSA
970
971 #if ALG_ECC
972         // If ECC is implemented ECDSA is required
973         case ALG_ECC_VALUE:
974             switch(scheme)
975             {
976                 // Support for ECDSA is required for ECC
977                 case ALG_ECDSA_VALUE:
978 #if ALG_ECDAA // ECDAA is optional
979                 case ALG_ECDAA_VALUE:
980 #endif
981 #if ALG_ECSCHNORR // Schnorr is also optional
982                 case ALG_ECSCHNORR_VALUE:
983 #endif
984 #if ALG_SM2 // SM2 is optional
985                 case ALG_SM2_VALUE:
986 #endif
987                     break;
988                 default:
989                     isSignScheme = FALSE;
990                     break;
991             }
992             break;
993 #endif // ALG_ECC
994         default:
995             isSignScheme = FALSE;
996             break;
997     }
998     return isSignScheme;
999 }
```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
1000 BOOL
1001 CryptIsAsymDecryptScheme(
1002     TPMI_ALG_PUBLIC          publicType,         // IN: Type of the object
1003     TPMI_ALG_ASYM_SCHEME     scheme              // IN: the scheme
1004     )
1005 {
1006     BOOL        isDecryptScheme = TRUE;
1007
1008     switch(publicType)
1009     {
1010 #if ALG_RSA
1011         case ALG_RSA_VALUE:
1012             switch(scheme)
1013             {
1014                 case ALG_RSAES_VALUE:
1015                 case ALG_OAEP_VALUE:
1016                     break;
1017                 default:
1018                     isDecryptScheme = FALSE;
1019                     break;
1020             }
1021             break;
```

```
#endif // ALG_RSA

#if ALG_ECC
        // If ECC is implemented ECDH is required
        case ALG_ECC_VALUE:
            switch(scheme)
            {
#if  !ALG_ECDH
#   error "ECDH is required for ECC"
#endif
                case ALG_ECDH_VALUE:
#if ALG_SM2
                case ALG_SM2_VALUE:
#endif
#if ALG_ECMQV
                case ALG_ECMQV_VALUE:
#endif
                    break;
                default:
                    isDecryptScheme = FALSE;
                    break;
            }
            break;
#endif // ALG_ECC
        default:
            isDecryptScheme = FALSE;
            break;
    }
    return isDecryptScheme;
}
```

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*

```
BOOL
CryptSelectSignScheme(
    OBJECT              *signObject,    // IN: signing key
    TPMT_SIG_SCHEME     *scheme         // IN/OUT: signing scheme
    )
{
    TPMT_SIG_SCHEME     *objectScheme;
    TPMT_PUBLIC         *publicArea;
    BOOL                 OK;

    // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
    // of the setting of scheme
    if(signObject == NULL)
```

```
1065        {
1066            OK = TRUE;
1067            scheme->scheme = ALG_NULL_VALUE;
1068            scheme->details.any.hashAlg = ALG_NULL_VALUE;
1069        }
1070        else
1071        {
1072            // assignment to save typing.
1073            publicArea = &signObject->publicArea;
1074
1075            // A symmetric cipher can be used to encrypt and decrypt but it can't
1076            // be used for signing
1077            if(publicArea->type == ALG_SYMCIPHER_VALUE)
1078                return FALSE;
1079            // Point to the scheme object
1080            if(CryptIsAsymAlgorithm(publicArea->type))
1081                objectScheme =
1082                (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
1083            else
1084                objectScheme =
1085                (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;
1086
1087        // If the object doesn't have a default scheme, then use the
1088        // input scheme.
1089            if(objectScheme->scheme == ALG_NULL_VALUE)
1090            {
1091                // Input and default can't both be NULL
1092                OK = (scheme->scheme != ALG_NULL_VALUE);
1093                // Assume that the scheme is compatible with the key. If not,
1094                // an error will be generated in the signing operation.
1095            }
1096            else if(scheme->scheme == ALG_NULL_VALUE)
1097            {
1098                // input scheme is NULL so use default
1099
1100                // First, check to see if the default requires that the caller
1101                // provided scheme data
1102                OK = !CryptIsSplitSign(objectScheme->scheme);
1103                if(OK)
1104                {
1105                    // The object has a scheme and the input is TPM_ALG_NULL so copy
1106                    // the object scheme as the final scheme. It is better to use a
1107                    // structure copy than a copy of the individual fields.
1108                    *scheme = *objectScheme;
1109                }
1110            }
1111            else
1112            {
1113                // Both input and object have scheme selectors
1114                // If the scheme and the hash are not the same then...
1115                // NOTE: the reason that there is no copy here is that the input
1116                // might contain extra data for a split signing scheme and that
1117                // data is not in the object so, it has to be preserved.
1118                OK = (objectScheme->scheme == scheme->scheme)
1119                    && (objectScheme->details.any.hashAlg
1120                        == scheme->details.any.hashAlg);
1121            }
1122        }
1123        return OK;
1124    }
```

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)

```
1125  TPM_RC
1126  CryptSign(
1127      OBJECT              *signKey,       // IN: signing key
1128      TPMT_SIG_SCHEME     *signScheme,    // IN: sign scheme.
1129      TPM2B_DIGEST        *digest,        // IN: The digest being signed
1130      TPMT_SIGNATURE      *signature      // OUT: signature
1131      )
1132  {
1133      TPM_RC               result = TPM_RC_SCHEME;
1134
1135      // Initialize signature scheme
1136      signature->sigAlg = signScheme->scheme;
1137
1138      // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1139      // then we are done
1140      if((signature->sigAlg == ALG_NULL_VALUE) || (signKey == NULL))
1141          return TPM_RC_SUCCESS;
1142
1143      // Initialize signature hash
1144      // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1145      // doesn't have a hashAlg member.
1146      signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1147
1148      // perform sign operation based on different key type
1149      switch(signKey->publicArea.type)
1150      {
1151  #if ALG_RSA
1152          case ALG_RSA_VALUE:
1153              result = CryptRsaSign(signature, signKey, digest, NULL);
1154              break;
1155  #endif // ALG_RSA
1156  #if ALG_ECC
1157          case ALG_ECC_VALUE:
1158              // The reason that signScheme is passed to CryptEccSign but not to the
1159              // other signing methods is that the signing for ECC may be split and
1160              // need the 'r' value that is in the scheme but not in the signature.
1161              result = CryptEccSign(signature, signKey, digest,
1162                                    (TPMT_ECC_SCHEME *)signScheme, NULL);
1163              break;
1164  #endif // ALG_ECC
1165          case ALG_KEYEDHASH_VALUE:
1166              result = CryptHmacSign(signature, signKey, digest);
1167              break;
1168          default:
1169              FAIL(FATAL_ERROR_INTERNAL);
1170              break;
1171      }
1172      return result;
1173  }
```

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

```
1174  TPM_RC
1175  CryptValidateSignature(
1176      TPMI_DH_OBJECT   keyHandle,     // IN: The handle of sign key
1177      TPM2B_DIGEST    *digest,        // IN: The digest being validated
1178      TPMT_SIGNATURE  *signature      // IN: signature
1179      )
1180  {
1181      // NOTE: HandleToObject will either return a pointer to a loaded object or
1182      // will assert. It will never return a non-valid value. This makes it save
1183      // to initialize 'publicArea' with the return value from HandleToObject()
1184      // without checking it first.
1185      OBJECT              *signObject = HandleToObject(keyHandle);
1186      TPMT_PUBLIC         *publicArea = &signObject->publicArea;
1187      TPM_RC               result = TPM_RC_SCHEME;
1188
1189      // The input unmarshaling should prevent any input signature from being
1190      // a NULL signature, but just in case
1191      if(signature->sigAlg == ALG_NULL_VALUE)
1192          return TPM_RC_SIGNATURE;
1193
1194      switch(publicArea->type)
1195      {
1196  #if ALG_RSA
1197          case ALG_RSA_VALUE:
1198          {
1199      //
1200              // Call RSA code to verify signature
1201              result = CryptRsaValidateSignature(signature, signObject, digest);
1202              break;
1203          }
1204  #endif // ALG_RSA
1205
1206  #if ALG_ECC
1207          case ALG_ECC_VALUE:
1208              result = CryptEccValidateSignature(signature, signObject, digest);
1209              break;
1210  #endif // ALG_ECC
1211
1212          case ALG_KEYEDHASH_VALUE:
1213              if(signObject->attributes.publicOnly)
1214                  result = TPM_RCS_HANDLE;
1215              else
1216                  result = CryptHMACVerifySignature(signObject, digest, signature);
1217              break;
1218          default:
1219              break;
1220      }
```

```
1221        return result;
1222    }
```

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().

```
1223    TPM_RC
1224    CryptGetTestResult(
1225        TPM2B_MAX_BUFFER    *outData        // OUT: test result data
1226        )
1227    {
1228        outData->t.size = 0;
1229        return TPM_RC_SUCCESS;
1230    }
```

10.2.6.6.17 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

```
1231    TPM_RC
1232    CryptValidateKeys(
1233        TPMT_PUBLIC      *publicArea,
1234        TPMT_SENSITIVE   *sensitive,
1235        TPM_RC            blamePublic,
1236        TPM_RC            blameSensitive
1237        )
1238    {
1239        TPM_RC               result;
1240        UINT16               keySizeInBytes;
1241        UINT16               digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1242        TPMU_PUBLIC_PARMS   *params = &publicArea->parameters;
1243        TPMU_PUBLIC_ID      *unique = &publicArea->unique;
1244
1245        if(sensitive != NULL)
1246        {
1247            // Make sure that the types of the public and sensitive are compatible
1248            if(publicArea->type != sensitive->sensitiveType)
1249                return TPM_RCS_TYPE + blameSensitive;
1250            // Make sure that the authValue is not bigger than allowed
1251            // If there is no name algorithm, then the size just needs to be less than
```

```
1252            // the maximum size of the buffer used for authorization. That size check
1253            // was made during unmarshaling of the sensitive area
1254            if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1255                return TPM_RCS_SIZE + blameSensitive;
1256        }
1257        switch(publicArea->type)
1258        {
1259    #if ALG_RSA
1260            case ALG_RSA_VALUE:
1261                keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1262
1263                // Regardless of whether there is a sensitive area, the public modulus
1264                // needs to have the correct size. Otherwise, it can't be used for
1265                // any public key operation nor can it be used to compute the private
1266                // exponent.
1267                // NOTE: This implementation only supports key sizes that are multiples
1268                // of 1024 bits which means that the MSb of the 0th byte will always be
1269                // SET in any prime and in the public modulus.
1270                if((unique->rsa.t.size != keySizeInBytes)
1271                    || (unique->rsa.t.buffer[0] < 0x80))
1272                    return TPM_RCS_KEY + blamePublic;
1273                if(params->rsaDetail.exponent != 0
1274                   && params->rsaDetail.exponent < 7)
1275                    return TPM_RCS_VALUE + blamePublic;
1276                if(sensitive != NULL)
1277                {
1278                    // If there is a sensitive area, it has to be the correct size
1279                    // including having the correct high order bit SET.
1280                    if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1281                       || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1282                        return TPM_RCS_KEY_SIZE + blameSensitive;
1283                }
1284                break;
1285    #endif
1286    #if ALG_ECC
1287            case ALG_ECC_VALUE:
1288            {
1289                TPMI_ECC_CURVE      curveId;
1290                curveId = params->eccDetail.curveID;
1291                keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1292                if(sensitive == NULL)
1293                {
1294                    // Validate the public key size
1295                    if(unique->ecc.x.t.size != keySizeInBytes
1296                       || unique->ecc.y.t.size != keySizeInBytes)
1297                        return TPM_RCS_KEY + blamePublic;
1298                    if(publicArea->nameAlg != ALG_NULL_VALUE)
1299                    {
1300                        if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
1301                            return TPM_RCS_ECC_POINT + blamePublic;
1302                    }
1303                }
1304                else
1305                {
1306                    // If the nameAlg is TPM_ALG_NULL, then only verify that the
1307                    // private part of the key is OK.
1308                    if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc,
1309                                                  curveId))
1310                        return TPM_RCS_KEY_SIZE;
1311                    if(publicArea->nameAlg != ALG_NULL_VALUE)
1312                    {
1313                    // Full key load, verify that the public point belongs to the
1314                    // private key.
1315                        TPMS_ECC_POINT          toCompare;
1316                        result = CryptEccPointMultiply(&toCompare, curveId, NULL,
1317                                                       &sensitive->sensitive.ecc,
```

```
                                                    NULL, NULL);
                    if(result != TPM_RC_SUCCESS)
                        return TPM_RCS_BINDING;
                    else
                    {
                    // Make sure that the private key generated the public key.
                    // The input values and the values produced by the point
                    // multiply may not be the same size so adjust the computed
                    // value to match the size of the input value by adding or
                    // removing zeros.
                        AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
                        AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
                        if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
                           || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
                            return TPM_RCS_BINDING;
                    }
                }
            }
            break;
        }
#endif
        default:
            // Checks for SYMCIPHER and KEYEDHASH are largely the same
            // If public area has a nameAlg, then validate the public area size
            // and if there is also a sensitive area, validate the binding

            // For consistency, if the object is public-only just make sure that
            // the unique field is consistent with the name algorithm
            if(sensitive == NULL)
            {
                if(unique->sym.t.size != digestSize)
                    return TPM_RCS_KEY + blamePublic;
            }
            else
            {
                // Make sure that the key size in the sensitive area is consistent.
                if(publicArea->type == ALG_SYMCIPHER_VALUE)
                {
                    result = CryptSymKeyValidate(&params->symDetail.sym,
                                                 &sensitive->sensitive.sym);
                    if(result != TPM_RC_SUCCESS)
                        return result + blameSensitive;
                }
                else
                {
                    // For a keyed hash object, the key has to be less than the
                    // smaller of the block size of the hash used in the scheme or
                    // 128 bytes. The worst case value is limited by the
                    // unmarshaling code so the only thing left to be checked is
                    // that it does not exceed the block size of the hash.
                    // by the hash algorithm of the scheme.
                    TPMT_KEYEDHASH_SCHEME       *scheme;
                    UINT16                       maxSize;
                    scheme = &params->keyedHashDetail.scheme;
                    if(scheme->scheme == ALG_XOR_VALUE)
                    {
                        maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
                    }
                    else if(scheme->scheme == ALG_HMAC_VALUE)
                    {
                        maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
                    }
                    else if(scheme->scheme == ALG_NULL_VALUE)
                    {
                        // Not signing or xor so must be a data block
                        maxSize = 128;
```

```
1384                    }
1385                    else
1386                        return TPM_RCS_SCHEME + blamePublic;
1387                    if(sensitive->sensitive.bits.t.size > maxSize)
1388                        return TPM_RCS_KEY_SIZE + blameSensitive;
1389                }
1390                // If there is a nameAlg, check the binding
1391                if(publicArea->nameAlg != ALG_NULL_VALUE)
1392                {
1393                    TPM2B_DIGEST            compare;
1394                    if(sensitive->seedValue.t.size != digestSize)
1395                        return TPM_RCS_KEY_SIZE + blameSensitive;
1396
1397                    CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1398                    if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1399                        return TPM_RC_BINDING;
1400                }
1401            }
1402            break;
1403        }
1404        // For a parent, need to check that the seedValue is the correct size for
1405        // protections. It should be at least half the size of the nameAlg
1406        if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
1407           && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
1408           && sensitive != NULL
1409           && publicArea->nameAlg != ALG_NULL_VALUE)
1410        {
1411            if((sensitive->seedValue.t.size < (digestSize / 2))
1412               || (sensitive->seedValue.t.size > digestSize))
1413                return TPM_RCS_SIZE + blameSensitive;
1414        }
1415        return TPM_RC_SUCCESS;
1416    }
```

10.2.6.6.18 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

```
1417    TPM_RC
1418    CryptSelectMac(
1419        TPMT_PUBLIC             *publicArea,
1420        TPMI_ALG_MAC_SCHEME     *inMac
1421    )
1422    {
1423        TPM_ALG_ID              macAlg = ALG_NULL_VALUE;
1424        switch(publicArea->type)
1425        {
1426            case ALG_KEYEDHASH_VALUE:
1427            {
1428                // Local value to keep lines from getting too long
1429                TPMT_KEYEDHASH_SCHEME   *scheme;
1430                scheme = &publicArea->parameters.keyedHashDetail.scheme;
1431                // Expect that the scheme is either HMAC or NULL
1432                if(scheme->scheme != ALG_NULL_VALUE)
1433                    macAlg = scheme->details.hmac.hashAlg;
1434                break;
1435            }
```

```
1436            case ALG_SYMCIPHER_VALUE:
1437            {
1438                TPMT_SYM_DEF_OBJECT     *scheme;
1439                scheme = &publicArea->parameters.symDetail.sym;
1440                // Expect that the scheme is either valid symmetric cipher or NULL
1441                if(scheme->algorithm != ALG_NULL_VALUE)
1442                    macAlg = scheme->mode.sym;
1443                break;
1444            }
1445            default:
1446                return TPM_RCS_TYPE;
1447        }
1448        // If the input value is not TPM_ALG_NULL ...
1449        if(*inMac != ALG_NULL_VALUE)
1450        {
1451            // ... then either the scheme in the key must be TPM_ALG_NULL or the input
1452            // value must match
1453            if((macAlg != ALG_NULL_VALUE) && (*inMac != macAlg))
1454                return TPM_RCS_VALUE;
1455        }
1456        else
1457        {
1458            // Since the input value is TPM_ALG_NULL, then the key value can't be
1459            // TPM_ALG_NULL
1460            if(macAlg == ALG_NULL_VALUE)
1461                return TPM_RCS_VALUE;
1462            *inMac = macAlg;
1463        }
1464        if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
1465            return TPM_RCS_SCHEME;
1466        return TPM_RC_SUCCESS;
1467    }
```

10.2.6.6.19 CryptMacIsValidForKey()

Check to see if the key type is compatible with the mac type

```
1468    BOOL
1469    CryptMacIsValidForKey(
1470        TPM_ALG_ID          keyType,
1471        TPM_ALG_ID          macAlg,
1472        BOOL                flag
1473    )
1474    {
1475        switch(keyType)
1476        {
1477            case ALG_KEYEDHASH_VALUE:
1478                return CryptHashIsValidAlg(macAlg, flag);
1479                break;
1480            case ALG_SYMCIPHER_VALUE:
1481                return CryptSmacIsValidAlg(macAlg, flag);
1482                break;
1483            default:
1484                break;
1485        }
1486        return FALSE;
1487    }
```

10.2.6.6.20 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.

```
1488    BOOL
1489    CryptSmacIsValidAlg(
1490        TPM_ALG_ID      alg,
1491        BOOL            FLAG        // IN: Indicates if TPM_ALG_NULL is valid
1492    )
1493    {
1494        switch (alg)
1495        {
1496    #if ALG_CMAC
1497            case ALG_CMAC_VALUE:
1498                return TRUE;
1499                break;
1500    #endif
1501            case ALG_NULL_VALUE:
1502                return FLAG;
1503                break;
1504            default:
1505                return FALSE;
1506        }
1507    }
```

10.2.6.6.21 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

```
1508    BOOL
1509    CryptSymModeIsValid(
1510        TPM_ALG_ID          mode,
1511        BOOL                flag
1512    )
1513    {
1514        switch(mode)
1515        {
1516    #if ALG_CTR
1517            case ALG_CTR_VALUE:
1518    #endif // ALG_CTR
1519    #if ALG_OFB
1520            case ALG_OFB_VALUE:
1521    #endif // ALG_OFB
1522    #if ALG_CBC
1523            case ALG_CBC_VALUE:
1524    #endif // ALG_CBC
1525    #if ALG_CFB
1526            case ALG_CFB_VALUE:
1527    #endif // ALG_CFB
1528    #if ALG_ECB
1529            case ALG_ECB_VALUE:
1530    #endif // ALG_ECB
1531                return TRUE;
1532            case ALG_NULL_VALUE:
1533                return flag;
1534                break;
1535            default:
1536                break;
1537        }
1538        return FALSE;
1539    }
```

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

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

10.2.7.2 Functions

10.2.7.2.1 RunSelfTest()

Local function to run self-test

```
static TPM_RC
CryptRunSelfTests(
    ALGORITHM_VECTOR    *toTest         // IN: the vector of the algorithms to test
    )
{
    TPM_ALG_ID           alg;

    // For each of the algorithms that are in the toTestVecor, need to run a
    // test
    for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
    {
        if(TEST_BIT(alg, *toTest))
        {
            TPM_RC          result = CryptTestAlgorithm(alg, toTest);
            if(result != TPM_RC_SUCCESS)
                return result;
        }
    }
    return TPM_RC_SUCCESS;
}
```

10.2.7.2.2 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

```
22  LIB_EXPORT
23  TPM_RC
24  CryptSelfTest(
25      TPMI_YES_NO      fullTest       // IN: if full test is required
26      )
27  {
28  #if SIMULATION
29      if(g_forceFailureMode)
30          FAIL(FATAL_ERROR_FORCED);
31  #endif
32
33      // If the caller requested a full test, then reset the to test vector so that
34      // all the tests will be run
35      if(fullTest == YES)
36      {
37          MemoryCopy(g_toTest,
38                     g_implementedAlgorithms,
39                     sizeof(g_toTest));
40      }
41      return CryptRunSelfTests(&g_toTest);
42  }
```

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

```
43  TPM_RC
44  CryptIncrementalSelfTest(
45      TPML_ALG            *toTest,        // IN: list of algorithms to be tested
46      TPML_ALG            *toDoList       // OUT: list of algorithms needing test
47      )
48  {
49      ALGORITHM_VECTOR     toTestVector = {0};
50      TPM_ALG_ID           alg;
51      UINT32               i;
52
53      pAssert(toTest != NULL && toDoList != NULL);
54      if(toTest->count > 0)
55      {
56          // Transcribe the toTest list into the toTestVector
57          for(i = 0; i < toTest->count; i++)
58          {
59              alg = toTest->algorithms[i];
60
61              // make sure that the algorithm value is not out of range
62              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.

```
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

```
#include "Tpm.h"
#include "OIDs.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.

```
#if ALG_ECC
#if USE_BN_ECC_DATA
#       define TO_ECC_64                           TO_CRYPT_WORD_64
#       define TO_ECC_56(a, b, c, d, e, f, g)      TO_ECC_64(0, a, b, c, d, e, f, g)
#       define TO_ECC_48(a, b, c, d, e, f)         TO_ECC_64(0, 0, a, b, c, d, e, f)
#       define TO_ECC_40(a, b, c, d, e)            TO_ECC_64(0, 0, 0, a, b, c, d, e)
#   if RADIX_BITS > 32
#       define TO_ECC_32(a, b, c, d)               TO_ECC_64(0, 0, 0, 0, a, b, c, d)
#       define TO_ECC_24(a, b, c)                  TO_ECC_64(0, 0, 0, 0, 0, a, b, c)
#       define TO_ECC_16(a, b)                     TO_ECC_64(0, 0, 0, 0, 0, 0, a, b)
#       define TO_ECC_8(a)                         TO_ECC_64(0, 0, 0, 0, 0, 0, 0, a)
#   else // RADIX_BITS == 32
#       define TO_ECC_32                           BIG_ENDIAN_BYTES_TO_UINT32
#       define TO_ECC_24(a, b, c)                  TO_ECC_32(0, a, b, c)
#       define TO_ECC_16(a, b)                     TO_ECC_32(0, 0, a, b)
#       define TO_ECC_8(a)                         TO_ECC_32(0, 0, 0, a)
#   endif
#else // TPM2B_
#   define TO_ECC_64(a, b, c, d, e, f, g, h) a, b, c, d, e, f, g, h
#   define TO_ECC_56(a, b, c, d, e, f, g)    a, b, c, d, e, f, g
#   define TO_ECC_48(a, b, c, d, e, f)       a, b, c, d, e, f
#   define TO_ECC_40(a, b, c, d, e)          a, b, c, d, e
#   define TO_ECC_32(a, b, c, d)             a, b, c, d
#   define TO_ECC_24(a, b, c)                a, b, c
#   define TO_ECC_16(a, b)                   a, b
#   define TO_ECC_8(a)                       a
#endif
#if USE_BN_ECC_DATA
#define BN_MIN_ALLOC(bytes)                                                          \
            (BYTES_TO_CRYPT_WORDS(bytes) == 0) ? 1 : BYTES_TO_CRYPT_WORDS(bytes)
# define ECC_CONST(NAME, bytes, initializer)                                         \
    const struct {                                                                   \
        crypt_uword_t   allocate, size, d[BN_MIN_ALLOC(bytes)];                      \
        } NAME = {BN_MIN_ALLOC(bytes), BYTES_TO_CRYPT_WORDS(bytes),{initializer}}
ECC_CONST(ECC_ZERO, 0, 0);
#else
# define ECC_CONST(NAME, bytes, initializer)                                         \
       const TPM2B_##bytes##_BYTE_VALUE NAME = {bytes, {initializer}}
```

Have to special case ECC_ZERO

```
TPM2B_BYTE_VALUE(1);
TPM2B_1_BYTE_VALUE ECC_ZERO = {1, {0}};

#endif

ECC_CONST(ECC_ONE, 1, 1);
#if !USE_BN_ECC_DATA
TPM2B_BYTE_VALUE(24);
#define TO_ECC_192(a, b, c)  a, b, c
TPM2B_BYTE_VALUE(28);
#define TO_ECC_224(a, b, c, d)   a, b, c, d
TPM2B_BYTE_VALUE(32);
```

```
53  #define TO_ECC_256(a, b, c, d)   a, b, c, d
54  TPM2B_BYTE_VALUE(48);
55  #define TO_ECC_384(a, b, c, d, e, f)     a, b, c, d, e, f
56  TPM2B_BYTE_VALUE(66);
57  #define TO_ECC_528(a, b, c, d, e, f, g, h, i)    a, b, c, d, e, f, g, h, i
58  TPM2B_BYTE_VALUE(80);
59  #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)     a, b, c, d, e, f, g, h, i, j
60  #else
61  #define TO_ECC_192(a, b, c)  c, b, a
62  #define TO_ECC_224(a, b, c, d)   d, c, b, a
63  #define TO_ECC_256(a, b, c, d)   d, c, b, a
64  #define TO_ECC_384(a, b, c, d, e, f)     f, e, d, c, b, a
65  #define TO_ECC_528(a, b, c, d, e, f, g, h, i)    i, h, g, f, e, d, c, b, a
66  #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)     j, i, h, g, f, e, d, c, b, a
67  #endif // !USE_BN_ECC_DATA
68  #if ECC_NIST_P192
69  ECC_CONST(NIST_P192_p, 24, TO_ECC_192(
70      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
71      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
72      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
73  ECC_CONST(NIST_P192_a, 24, TO_ECC_192(
74      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
75      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
76      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
77  ECC_CONST(NIST_P192_b, 24, TO_ECC_192(
78      TO_ECC_64(0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7),
79      TO_ECC_64(0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49),
80      TO_ECC_64(0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1)));
81  ECC_CONST(NIST_P192_gX, 24, TO_ECC_192(
82      TO_ECC_64(0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6),
83      TO_ECC_64(0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00),
84      TO_ECC_64(0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12)));
85  ECC_CONST(NIST_P192_gY, 24, TO_ECC_192(
86      TO_ECC_64(0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78),
87      TO_ECC_64(0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5),
88      TO_ECC_64(0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11)));
89  ECC_CONST(NIST_P192_n, 24, TO_ECC_192(
90      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
91      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36),
92      TO_ECC_64(0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31)));
93  #define NIST_P192_h          ECC_ONE
94  #define NIST_P192_gZ         ECC_ONE
95  #if USE_BN_ECC_DATA
96      const ECC_CURVE_DATA NIST_P192 = {
97          (bigNum)&NIST_P192_p, (bigNum)&NIST_P192_n, (bigNum)&NIST_P192_h,
98          (bigNum)&NIST_P192_a, (bigNum)&NIST_P192_b,
99          {(bigNum)&NIST_P192_gX, (bigNum)&NIST_P192_gY, (bigNum)&NIST_P192_gZ}};
100 #else
101     const ECC_CURVE_DATA NIST_P192 = {
102         &NIST_P192_p.b, &NIST_P192_n.b, &NIST_P192_h.b,
103         &NIST_P192_a.b, &NIST_P192_b.b,
104         {&NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_gZ.b}};
105
106 #endif // USE_BN_ECC_DATA
107
108 #endif // ECC_NIST_P192
109
110
111 #if ECC_NIST_P224
112 ECC_CONST(NIST_P224_p, 28, TO_ECC_224(
113     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
114     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
115     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
116     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01)));
117 ECC_CONST(NIST_P224_a, 28, TO_ECC_224(
```

```
118         TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
119         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
120         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
121         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE)));
122     ECC_CONST(NIST_P224_b, 28, TO_ECC_224(
123         TO_ECC_32(0xB4, 0x05, 0x0A, 0x85),
124         TO_ECC_64(0x0C, 0x04, 0xB3, 0xAB, 0xF5, 0x41, 0x32, 0x56),
125         TO_ECC_64(0x50, 0x44, 0xB0, 0xB7, 0xD7, 0xBF, 0xD8, 0xBA),
126         TO_ECC_64(0x27, 0x0B, 0x39, 0x43, 0x23, 0x55, 0xFF, 0xB4)));
127     ECC_CONST(NIST_P224_gX, 28, TO_ECC_224(
128         TO_ECC_32(0xB7, 0x0E, 0x0C, 0xBD),
129         TO_ECC_64(0x6B, 0xB4, 0xBF, 0x7F, 0x32, 0x13, 0x90, 0xB9),
130         TO_ECC_64(0x4A, 0x03, 0xC1, 0xD3, 0x56, 0xC2, 0x11, 0x22),
131         TO_ECC_64(0x34, 0x32, 0x80, 0xD6, 0x11, 0x5C, 0x1D, 0x21)));
132     ECC_CONST(NIST_P224_gY, 28, TO_ECC_224(
133         TO_ECC_32(0xBD, 0x37, 0x63, 0x88),
134         TO_ECC_64(0xB5, 0xF7, 0x23, 0xFB, 0x4C, 0x22, 0xDF, 0xE6),
135         TO_ECC_64(0xCD, 0x43, 0x75, 0xA0, 0x5A, 0x07, 0x47, 0x64),
136         TO_ECC_64(0x44, 0xD5, 0x81, 0x99, 0x85, 0x00, 0x7E, 0x34)));
137     ECC_CONST(NIST_P224_n, 28, TO_ECC_224(
138         TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
139         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
140         TO_ECC_64(0xFF, 0xFF, 0x16, 0xA2, 0xE0, 0xB8, 0xF0, 0x3E),
141         TO_ECC_64(0x13, 0xDD, 0x29, 0x45, 0x5C, 0x5C, 0x2A, 0x3D)));
142     #define NIST_P224_h          ECC_ONE
143     #define NIST_P224_gZ         ECC_ONE
144     #if USE_BN_ECC_DATA
145         const ECC_CURVE_DATA NIST_P224 = {
146             (bigNum)&NIST_P224_p, (bigNum)&NIST_P224_n, (bigNum)&NIST_P224_h,
147             (bigNum)&NIST_P224_a, (bigNum)&NIST_P224_b,
148             {(bigNum)&NIST_P224_gX, (bigNum)&NIST_P224_gY, (bigNum)&NIST_P224_gZ}};
149     #else
150         const ECC_CURVE_DATA NIST_P224 = {
151             &NIST_P224_p.b, &NIST_P224_n.b, &NIST_P224_h.b,
152             &NIST_P224_a.b, &NIST_P224_b.b,
153             {&NIST_P224_gX.b, &NIST_P224_gY.b, &NIST_P224_gZ.b}};
154
155     #endif // USE_BN_ECC_DATA
156
157     #endif // ECC_NIST_P224
158
159
160     #if ECC_NIST_P256
161     ECC_CONST(NIST_P256_p, 32, TO_ECC_256(
162         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
163         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
164         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
165         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
166     ECC_CONST(NIST_P256_a, 32, TO_ECC_256(
167         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
168         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
169         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
170         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
171     ECC_CONST(NIST_P256_b, 32, TO_ECC_256(
172         TO_ECC_64(0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7),
173         TO_ECC_64(0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC),
174         TO_ECC_64(0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6),
175         TO_ECC_64(0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B)));
176     ECC_CONST(NIST_P256_gX, 32, TO_ECC_256(
177         TO_ECC_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47),
178         TO_ECC_64(0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
179         TO_ECC_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
180         TO_ECC_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96)));
181     ECC_CONST(NIST_P256_gY, 32, TO_ECC_256(
182         TO_ECC_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B),
```

```
183         TO_ECC_64(0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16),
184         TO_ECC_64(0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
185         TO_ECC_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5)));
186     ECC_CONST(NIST_P256_n, 32, TO_ECC_256(
187         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
188         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
189         TO_ECC_64(0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84),
190         TO_ECC_64(0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51)));
191     #define NIST_P256_h         ECC_ONE
192     #define NIST_P256_gZ        ECC_ONE
193     #if USE_BN_ECC_DATA
194         const ECC_CURVE_DATA NIST_P256 = {
195             (bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
196             (bigNum)&NIST_P256_a, (bigNum)&NIST_P256_b,
197             {(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&NIST_P256_gZ}};
198     #else
199         const ECC_CURVE_DATA NIST_P256 = {
200             &NIST_P256_p.b, &NIST_P256_n.b, &NIST_P256_h.b,
201             &NIST_P256_a.b, &NIST_P256_b.b,
202             {&NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_gZ.b}};
203
204     #endif // USE_BN_ECC_DATA
205
206     #endif // ECC_NIST_P256
207
208
209     #if ECC_NIST_P384
210     ECC_CONST(NIST_P384_p, 48, TO_ECC_384(
211         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
212         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
213         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
214         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
215         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
216         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF)));
217     ECC_CONST(NIST_P384_a, 48, TO_ECC_384(
218         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
219         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
220         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
221         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
222         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
223         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC)));
224     ECC_CONST(NIST_P384_b, 48, TO_ECC_384(
225         TO_ECC_64(0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4),
226         TO_ECC_64(0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19),
227         TO_ECC_64(0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12),
228         TO_ECC_64(0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A),
229         TO_ECC_64(0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D),
230         TO_ECC_64(0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF)));
231     ECC_CONST(NIST_P384_gX, 48, TO_ECC_384(
232         TO_ECC_64(0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37),
233         TO_ECC_64(0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74),
234         TO_ECC_64(0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98),
235         TO_ECC_64(0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38),
236         TO_ECC_64(0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C),
237         TO_ECC_64(0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7)));
238     ECC_CONST(NIST_P384_gY, 48, TO_ECC_384(
239         TO_ECC_64(0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F),
240         TO_ECC_64(0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29),
241         TO_ECC_64(0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C),
242         TO_ECC_64(0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0),
243         TO_ECC_64(0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D),
244         TO_ECC_64(0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F)));
245     ECC_CONST(NIST_P384_n, 48, TO_ECC_384(
246         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
247         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
```

```
248         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
249         TO_ECC_64(0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF),
250         TO_ECC_64(0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A),
251         TO_ECC_64(0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73)));
252     #define NIST_P384_h          ECC_ONE
253     #define NIST_P384_gZ         ECC_ONE
254     #if USE_BN_ECC_DATA
255         const ECC_CURVE_DATA NIST_P384 = {
256             (bigNum)&NIST_P384_p, (bigNum)&NIST_P384_n, (bigNum)&NIST_P384_h,
257             (bigNum)&NIST_P384_a, (bigNum)&NIST_P384_b,
258             {(bigNum)&NIST_P384_gX, (bigNum)&NIST_P384_gY, (bigNum)&NIST_P384_gZ}};
259     #else
260         const ECC_CURVE_DATA NIST_P384 = {
261             &NIST_P384_p.b, &NIST_P384_n.b, &NIST_P384_h.b,
262             &NIST_P384_a.b, &NIST_P384_b.b,
263             {&NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_gZ.b}};
264
265     #endif // USE_BN_ECC_DATA
266
267     #endif // ECC_NIST_P384
268
269
270     #if ECC_NIST_P521
271     ECC_CONST(NIST_P521_p, 66, TO_ECC_528(
272         TO_ECC_16(0x01, 0xFF),
273         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
274         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
275         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
276         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
277         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
278         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
279         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
280         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
281     ECC_CONST(NIST_P521_a, 66, TO_ECC_528(
282         TO_ECC_16(0x01, 0xFF),
283         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
284         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
285         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
286         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
287         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
288         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
289         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
290         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
291     ECC_CONST(NIST_P521_b, 66, TO_ECC_528(
292         TO_ECC_16(0x00, 0x51),
293         TO_ECC_64(0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C, 0x9A, 0x1F),
294         TO_ECC_64(0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85, 0x40, 0xEE),
295         TO_ECC_64(0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3, 0x15, 0xF3),
296         TO_ECC_64(0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1, 0x09, 0xE1),
297         TO_ECC_64(0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E, 0x93, 0x7B),
298         TO_ECC_64(0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1, 0xBF, 0x07),
299         TO_ECC_64(0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C, 0x34, 0xF1),
300         TO_ECC_64(0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50, 0x3F, 0x00)));
301     ECC_CONST(NIST_P521_gX, 66, TO_ECC_528(
302         TO_ECC_16(0x00, 0xC6),
303         TO_ECC_64(0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04, 0xE9, 0xCD),
304         TO_ECC_64(0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95, 0xB4, 0x42),
305         TO_ECC_64(0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F, 0xB5, 0x21),
306         TO_ECC_64(0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D, 0x3D, 0xBA),
307         TO_ECC_64(0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7, 0x59, 0x28),
308         TO_ECC_64(0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF, 0xA8, 0xDE),
309         TO_ECC_64(0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A, 0x42, 0x9B),
310         TO_ECC_64(0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5, 0xBD, 0x66)));
311     ECC_CONST(NIST_P521_gY, 66, TO_ECC_528(
312         TO_ECC_16(0x01, 0x18),
```

```
313         TO_ECC_64(0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B, 0xC0, 0x04),
314         TO_ECC_64(0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D, 0x1B, 0xD9),
315         TO_ECC_64(0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B, 0x44, 0x68),
316         TO_ECC_64(0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E, 0x66, 0x2C),
317         TO_ECC_64(0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4, 0x26, 0x40),
318         TO_ECC_64(0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD, 0x07, 0x61),
319         TO_ECC_64(0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72, 0xC2, 0x40),
320         TO_ECC_64(0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1, 0x66, 0x50)));
321     ECC_CONST(NIST_P521_n, 66, TO_ECC_528(
322         TO_ECC_16(0x01, 0xFF),
323         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
324         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
325         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
326         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFA),
327         TO_ECC_64(0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F, 0x96, 0x6B),
328         TO_ECC_64(0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09, 0xA5, 0xD0),
329         TO_ECC_64(0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C, 0x47, 0xAE),
330         TO_ECC_64(0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38, 0x64, 0x09)));
331     #define NIST_P521_h          ECC_ONE
332     #define NIST_P521_gZ         ECC_ONE
333     #if USE_BN_ECC_DATA
334         const ECC_CURVE_DATA NIST_P521 = {
335             (bigNum)&NIST_P521_p, (bigNum)&NIST_P521_n, (bigNum)&NIST_P521_h,
336             (bigNum)&NIST_P521_a, (bigNum)&NIST_P521_b,
337             {(bigNum)&NIST_P521_gX, (bigNum)&NIST_P521_gY, (bigNum)&NIST_P521_gZ}};
338     #else
339         const ECC_CURVE_DATA NIST_P521 = {
340             &NIST_P521_p.b, &NIST_P521_n.b, &NIST_P521_h.b,
341             &NIST_P521_a.b, &NIST_P521_b.b,
342             {&NIST_P521_gX.b, &NIST_P521_gY.b, &NIST_P521_gZ.b}};
343
344     #endif // USE_BN_ECC_DATA
345
346     #endif // ECC_NIST_P521
347
348
349     #if ECC_BN_P256
350     ECC_CONST(BN_P256_p, 32, TO_ECC_256(
351         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
352         TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9F),
353         TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x98, 0x0A, 0x82),
354         TO_ECC_64(0xD3, 0x29, 0x2D, 0xDB, 0xAE, 0xD3, 0x30, 0x13)));
355     #define BN_P256_a            ECC_ZERO
356     ECC_CONST(BN_P256_b, 1, TO_ECC_8(3));
357     #define BN_P256_gX           ECC_ONE
358     ECC_CONST(BN_P256_gY, 1, TO_ECC_8(2));
359     ECC_CONST(BN_P256_n, 32, TO_ECC_256(
360         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
361         TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9E),
362         TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x99, 0x92, 0x1A),
363         TO_ECC_64(0xF6, 0x2D, 0x53, 0x6C, 0xD1, 0x0B, 0x50, 0x0D)));
364     #define BN_P256_h            ECC_ONE
365     #define BN_P256_gZ           ECC_ONE
366     #if USE_BN_ECC_DATA
367         const ECC_CURVE_DATA BN_P256 = {
368             (bigNum)&BN_P256_p, (bigNum)&BN_P256_n, (bigNum)&BN_P256_h,
369             (bigNum)&BN_P256_a, (bigNum)&BN_P256_b,
370             {(bigNum)&BN_P256_gX, (bigNum)&BN_P256_gY, (bigNum)&BN_P256_gZ}};
371     #else
372         const ECC_CURVE_DATA BN_P256 = {
373             &BN_P256_p.b, &BN_P256_n.b, &BN_P256_h.b,
374             &BN_P256_a.b, &BN_P256_b.b,
375             {&BN_P256_gX.b, &BN_P256_gY.b, &BN_P256_gZ.b}};
376
377     #endif // USE_BN_ECC_DATA
```

```
378
379 #endif // ECC_BN_P256
380
381
382 #if ECC_BN_P638
383 ECC_CONST(BN_P638_p, 80, TO_ECC_640(
384     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
385     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
386     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
387     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
388     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
389     TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
390     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
391     TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
392     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
393     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67)));
394 #define BN_P638_a           ECC_ZERO
395 ECC_CONST(BN_P638_b, 2, TO_ECC_16(0x01,0x01));
396 ECC_CONST(BN_P638_gX, 80, TO_ECC_640(
397     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
398     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
399     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
400     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
401     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
402     TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
403     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
404     TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
405     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
406     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66)));
407 ECC_CONST(BN_P638_gY, 1, TO_ECC_8(0x10));
408 ECC_CONST(BN_P638_n, 80, TO_ECC_640(
409     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
410     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
411     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
412     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
413     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
414     TO_ECC_64(0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55),
415     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0),
416     TO_ECC_64(0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9),
417     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0),
418     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61)));
419 #define BN_P638_h           ECC_ONE
420 #define BN_P638_gZ          ECC_ONE
421 #if USE_BN_ECC_DATA
422     const ECC_CURVE_DATA BN_P638 = {
423         (bigNum)&BN_P638_p, (bigNum)&BN_P638_n, (bigNum)&BN_P638_h,
424         (bigNum)&BN_P638_a, (bigNum)&BN_P638_b,
425         {(bigNum)&BN_P638_gX, (bigNum)&BN_P638_gY, (bigNum)&BN_P638_gZ}};
426 #else
427     const ECC_CURVE_DATA BN_P638 = {
428         &BN_P638_p.b, &BN_P638_n.b, &BN_P638_h.b,
429         &BN_P638_a.b, &BN_P638_b.b,
430         {&BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_gZ.b}};
431
432 #endif // USE_BN_ECC_DATA
433
434 #endif // ECC_BN_P638
435
436
437 #if ECC_SM2_P256
438 ECC_CONST(SM2_P256_p, 32, TO_ECC_256(
439     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
440     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
441     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
442     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
```

```
443   ECC_CONST(SM2_P256_a, 32, TO_ECC_256(
444       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
445       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
446       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
447       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
448   ECC_CONST(SM2_P256_b, 32, TO_ECC_256(
449       TO_ECC_64(0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34),
450       TO_ECC_64(0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7),
451       TO_ECC_64(0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92),
452       TO_ECC_64(0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93)));
453   ECC_CONST(SM2_P256_gX, 32, TO_ECC_256(
454       TO_ECC_64(0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19),
455       TO_ECC_64(0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94),
456       TO_ECC_64(0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1),
457       TO_ECC_64(0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
458   ECC_CONST(SM2_P256_gY, 32, TO_ECC_256(
459       TO_ECC_64(0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C),
460       TO_ECC_64(0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53),
461       TO_ECC_64(0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40),
462       TO_ECC_64(0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0)));
463   ECC_CONST(SM2_P256_n, 32, TO_ECC_256(
464       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
465       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
466       TO_ECC_64(0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B),
467       TO_ECC_64(0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23)));
468   #define SM2_P256_h          ECC_ONE
469   #define SM2_P256_gZ         ECC_ONE
470   #if USE_BN_ECC_DATA
471       const ECC_CURVE_DATA SM2_P256 = {
472           (bigNum)&SM2_P256_p, (bigNum)&SM2_P256_n, (bigNum)&SM2_P256_h,
473           (bigNum)&SM2_P256_a, (bigNum)&SM2_P256_b,
474           {(bigNum)&SM2_P256_gX, (bigNum)&SM2_P256_gY, (bigNum)&SM2_P256_gZ}};
475   #else
476       const ECC_CURVE_DATA SM2_P256 = {
477           &SM2_P256_p.b, &SM2_P256_n.b, &SM2_P256_h.b,
478           &SM2_P256_a.b, &SM2_P256_b.b,
479           {&SM2_P256_gX.b, &SM2_P256_gY.b, &SM2_P256_gZ.b}};
480
481   #endif // USE_BN_ECC_DATA
482
483   #endif // ECC_SM2_P256
484
485
486   #define comma
487   const ECC_CURVE   eccCurves[] = {
488   #if ECC_NIST_P192
489       comma
490       {TPM_ECC_NIST_P192,
491       192,
492       {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
493       {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
494       &NIST_P192,
495       OID_ECC_NIST_P192
496       CURVE_NAME("NIST_P192")}
497   #   undef comma
498   #   define comma ,
499   #endif // ECC_NIST_P192
500   #if ECC_NIST_P224
501       comma
502       {TPM_ECC_NIST_P224,
503       224,
504       {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
505       {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
506       &NIST_P224,
507       OID_ECC_NIST_P224
```

```
508         CURVE_NAME("NIST_P224")}
509     #   undef comma
510     #   define comma ,
511     #endif // ECC_NIST_P224
512     #if ECC_NIST_P256
513         comma
514         {TPM_ECC_NIST_P256,
515         256,
516         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
517         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
518         &NIST_P256,
519         OID_ECC_NIST_P256
520         CURVE_NAME("NIST_P256")}
521     #   undef comma
522     #   define comma ,
523     #endif // ECC_NIST_P256
524     #if ECC_NIST_P384
525         comma
526         {TPM_ECC_NIST_P384,
527         384,
528         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA384_VALUE}}},
529         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
530         &NIST_P384,
531         OID_ECC_NIST_P384
532         CURVE_NAME("NIST_P384")}
533     #   undef comma
534     #   define comma ,
535     #endif // ECC_NIST_P384
536     #if ECC_NIST_P521
537         comma
538         {TPM_ECC_NIST_P521,
539         521,
540         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA512_VALUE}}},
541         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
542         &NIST_P521,
543         OID_ECC_NIST_P521
544         CURVE_NAME("NIST_P521")}
545     #   undef comma
546     #   define comma ,
547     #endif // ECC_NIST_P521
548     #if ECC_BN_P256
549         comma
550         {TPM_ECC_BN_P256,
551         256,
552         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
553         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
554         &BN_P256,
555         OID_ECC_BN_P256
556         CURVE_NAME("BN_P256")}
557     #   undef comma
558     #   define comma ,
559     #endif // ECC_BN_P256
560     #if ECC_BN_P638
561         comma
562         {TPM_ECC_BN_P638,
563         638,
564         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
565         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
566         &BN_P638,
567         OID_ECC_BN_P638
568         CURVE_NAME("BN_P638")}
569     #   undef comma
570     #   define comma ,
571     #endif // ECC_BN_P638
572     #if ECC_SM2_P256
573         comma
```

```
574         {TPM_ECC_SM2_P256,
575         256,
576         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SM3_256_VALUE}}},
577         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
578         &SM2_P256,
579         OID_ECC_SM2_P256
580         CURVE_NAME("SM2_P256")}
581     #   undef comma
582     #   define comma ,
583     #endif // ECC_SM2_P256
584     };
585     #endif // TPM_ALG_ECC
```

10.2.9 CryptDes.c

10.2.9.1 Introduction

This file contains the extra functions required for TDES.

10.2.9.2 Includes, Defines, and Typedefs

```
#include "Tpm.h"
#if ALG_TDES
#define DES_NUM_WEAK 64
const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
    0x0101010101010101ULL, 0xFEFEFEFEFEFEFEFEULL,
    0xE0E0E0E0F1F1F1F1ULL, 0x1F1F1F1F0E0E0E0EULL,
    0x011F011F010E010EULL, 0x1F011F010E010E01ULL,
    0x01E001E001F101F1ULL, 0xE001E001F101F101ULL,
    0x01FE01FE01FE01FEULL, 0xFE01FE01FE01FE01ULL,
    0x1FE01FE00EF10EF1ULL, 0xE01FE01FF10EF10EULL,
    0x1FFE1FFE0EFE0EFEULL, 0xFE1FFE1FFE0EFE0EULL,
    0xE0FEE0FEF1FEF1FEULL, 0xFEE0FEE0FEF1FEF1ULL,
    0x01011F1F01010E0EULL, 0x1F1F01010E0E0101ULL,
    0xE0E01F1FF1F10E0EULL, 0x0101E0E00101F1F1ULL,
    0x1F1FE0E00E0EF1F1ULL, 0xE0E0FEFEF1F1FEFEULL,
    0x0101FEFE0101FEFEULL, 0x1F1FFEFE0E0EFEFEULL,
    0xE0FE011FF1FE010EULL, 0x011F1F01010E0E01ULL,
    0x1FE001FE0EF101FEULL, 0xE0FE1F01F1FE0E01ULL,
    0x011FE0FE010EF1FEULL, 0x1FE0E01F0EF1F10EULL,
    0xE0FEFEE0F1FEFEF1ULL, 0x011FFEE0010EFEF1ULL,
    0x1FE0FE010EF1FE01ULL, 0xFE0101FEFE0101FEULL,
    0x01E01FFE01F10EFEULL, 0x1FFE01E00EFE01F1ULL,
    0xFE011FE0FE010EF1ULL, 0xFE01E01FFE01F10EULL,
    0x1FFEE0010EFEF101ULL, 0xFE1F01E0FE0E01F1ULL,
    0x01E0E00101F1F101ULL, 0x1FFEFE1F0EFEFE0EULL,
    0xFE1FE001FE0EF101ULL, 0x01E0FE1F01F1FE0EULL,
    0xE00101E0F10101F1ULL, 0xFE1F1FFEFE0E0EFEULL,
    0x01FE1FE001FE0EF1ULL, 0xE0011FFEF1010EFEULL,
    0xFEE0011FFEF1010EULL, 0x01FEE01F01FEF10EULL,
    0xE001FE1FF101FE0EULL, 0xFEE01F01FEF10E01ULL,
    0x01FEFE0101FEFE01ULL, 0xE01F01FEF10E01FEULL,
    0xFEE0E0FEFEF1F1FEULL, 0x1F01011F0E01010EULL,
    0xE01F1FE0F10E0EF1ULL, 0xFEFE0101FEFE0101ULL,
    0x1F01E0FE0E01F1FEULL, 0xE01FFE01F10EFE01ULL,
    0xFEFE1F1FFEFE0E0EULL, 0x1F01FEE00E01FEF1ULL,
    0xE0E00101F1F10101ULL, 0xFEFEE0E0FEFEF1F1ULL};

//*** CryptSetOddByteParity()
// This function sets the per byte parity of a 64-bit value. The least-significant
// bit is of each byte is replaced with the odd parity of the other 7 bits in the
// byte. With odd parity, no byte will ever be 0x00.
UINT64
CryptSetOddByteParity(
    UINT64          k
    )
{
#define PMASK 0x0101010101010101ULL
    UINT64          out;
    k |= PMASK;     // set the parity bit
    out = k;
    k ^= k >> 4;
    k ^= k >> 2;
    k ^= k >> 1;
```

```
54      k &= PMASK;     // odd parity extracted
55      out ^= k;       // out is now even parity because parity bit was already set
56      out ^= PMASK;   // out is now even parity
57      return out;
58  }
```

10.2.9.2.1 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

```
59  static BOOL
60  CryptDesIsWeakKey(
61      UINT64              k
62      )
63  {
64      int                 i;
65  //
66      for(i = 0; i < DES_NUM_WEAK; i++)
67      {
68          if(k == DesWeakKeys[i])
69              return TRUE;
70      }
71      return FALSE;
72  }
```

10.2.9.2.2 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.

```
73  BOOL
74  CryptDesValidateKey(
75      TPM2B_SYM_KEY       *desKey     // IN: key to validate
76      )
77  {
78      UINT64               k[3];
79      int                  i;
80      int                  keys = (desKey->t.size + 7) / 8;
81      BYTE                *pk = desKey->t.buffer;
82      BOOL                 ok;
83  //
84      // Note: 'keys' is the number of keys, not the maximum index for 'k'
85      ok = ((keys == 2) || (keys == 3)) && ((desKey->t.size % 8) == 0);
86      for(i = 0; ok && i < keys; pk += 8, i++)
87      {
88          k[i] = CryptSetOddByteParity(BYTE_ARRAY_TO_UINT64(pk));
89          ok = !CryptDesIsWeakKey(k[i]);
90      }
91      ok = ok && k[0] != k[1];
92      if(keys == 3)
93          ok = ok && k[1] != k[2];
94      return ok;
95  }
```

10.2.9.2.3 CryptGenerateKeyDes()

This function is used to create a DES key of the appropriate size. The key will have odd parity in the bytes.

```
 96  TPM_RC
 97  CryptGenerateKeyDes(
 98      TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
 99                                                  //     for the new key.
100      TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
101      RAND_STATE              *rand               // IN: the "entropy" source for
102      )
103  {
104
105      // Assume that the publicArea key size has been validated and is a supported
106      // number of bits.
107      sensitive->sensitive.sym.t.size =
108              BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
109      do
110      {
111          BYTE                    *pK = sensitive->sensitive.sym.t.buffer;
112          int                      i = (sensitive->sensitive.sym.t.size + 7) / 8;
113  // Use the random number generator to generate the required number of bits
114          if(DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
115              return TPM_RC_NO_RESULT;
116          for(; i > 0; pK += 8, i--)
117          {
118              UINT64      k = BYTE_ARRAY_TO_UINT64(pK);
119              k = CryptSetOddByteParity(k);
120              UINT64_TO_BYTE_ARRAY(k, pK);
121          }
122      } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
123      return TPM_RC_SUCCESS;
124  }
125  #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 from 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  }
```

10.2.10.2.7 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);
```

```
72      return (curve != NULL) ? curve->curveData : NULL;
73  }
```

10.2.11.2.7 CryptEccGetOID()

```
74  const BYTE *
75  CryptEccGetOID(
76      TPM_ECC_CURVE       curveId
77  )
78  {
79      const ECC_CURVE         *curve = CryptEccGetParametersByCurveId(curveId);
80      return (curve != NULL) ? curve->OID : NULL;
81  }
```

10.2.11.2.8 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.

```
82  LIB_EXPORT TPM_ECC_CURVE
83  CryptEccGetCurveByIndex(
84      UINT16              i
85      )
86  {
87      if(i >= ECC_CURVE_COUNT)
88          return TPM_ECC_NONE;
89      return eccCurves[i].curveId;
90  }
```

10.2.11.2.9 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

```
91  LIB_EXPORT BOOL
92  CryptEccGetParameter(
93      TPM2B_ECC_PARAMETER     *out,       // OUT: place to put parameter
94      char                     p,         // IN: the parameter selector
95      TPM_ECC_CURVE            curveId    // IN: the curve id
96      )
97  {
98      const ECC_CURVE_DATA    *curve = GetCurveData(curveId);
99      bigConst                 parameter = NULL;
100
101     if(curve != NULL)
102     {
103         switch(p)
104         {
105             case 'p':
106                 parameter = CurveGetPrime(curve);
107                 break;
108             case 'n':
109                 parameter = CurveGetOrder(curve);
110                 break;
```

```
111                 case 'a':
112                     parameter = CurveGet_a(curve);
113                     break;
114                 case 'b':
115                     parameter = CurveGet_b(curve);
116                     break;
117                 case 'x':
118                     parameter = CurveGetGx(curve);
119                     break;
120                 case 'y':
121                     parameter = CurveGetGy(curve);
122                     break;
123                 case 'h':
124                     parameter = CurveGetCofactor(curve);
125                     break;
126                 default:
127                     FAIL(FATAL_ERROR_INTERNAL);
128                     break;
129             }
130         }
131         // If not debugging and we get here with parameter still NULL, had better
132         // not try to convert so just return FALSE instead.
133         return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;
134     }
```

10.2.11.2.10 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

```
135     TPMI_YES_NO
136     CryptCapGetECCCurve(
137         TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
138         UINT32           maxCount,      // IN: count of returned curves
139         TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
140         )
141     {
142         TPMI_YES_NO       more = NO;
143         UINT16            i;
144         UINT32            count = ECC_CURVE_COUNT;
145         TPM_ECC_CURVE     curve;
146
147         // Initialize output property list
148         curveList->count = 0;
149
150         // The maximum count of curves we may return is MAX_ECC_CURVES
151         if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
152
153         // Scan the eccCurveValues array
154         for(i = 0; i < count; i++)
155         {
156             curve = CryptEccGetCurveByIndex(i);
157             // If curveID is less than the starting curveID, skip it
158             if(curve < curveID)
159                 continue;
160             if(curveList->count < maxCount)
161             {
162                 // If we have not filled up the return list, add more curves to
163                 // it
```

```
164             curveList->eccCurves[curveList->count] = curve;
165             curveList->count++;
166         }
167         else
168         {
169             // If the return list is full but we still have curves
170             // available, report this and stop iterating
171             more = YES;
172             break;
173         }
174     }
175     return more;
176 }
```

10.2.11.2.11 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
177 const TPMT_ECC_SCHEME *
178 CryptGetCurveSignScheme(
179     TPM_ECC_CURVE    curveId        // IN: The curve selector
180     )
181 {
182     const ECC_CURVE         *curve = CryptEccGetParametersByCurveId(curveId);
183 
184     if(curve != NULL)
185         return &(curve->sign);
186     else
187         return NULL;
188 }
```

10.2.11.2.12 CryptGenerateR()

This function computes the commit random value for a split signing scheme.

If *c* is NULL, it indicates that *r* is being generated for TPM2_Commit(). If *c* is not NULL, the TPM will validate that the *gr.commitArray* bit associated with the input value of *c* is SET. If not, the TPM returns FALSE and no *r* value is generated.

Return Value	Meaning
TRUE(1)	r value computed
FALSE(0)	no r value computed

```
189 BOOL
190 CryptGenerateR(
191     TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
192     UINT16                  *c,             // IN/OUT: count value.
193     TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
194     TPM2B_NAME              *name           // IN: optional name of a key to
195                                             //     associate with 'r'
196     )
197 {
198     // This holds the marshaled g_commitCounter.
199     TPM2B_TYPE(8B, 8);
200     TPM2B_8B                 cntr = {{8,{0}}};
201     UINT32                   iterations;
202     TPM2B_ECC_PARAMETER      n;
203     UINT64                   currentCount = gr.commitCounter;
204     UINT16                   t1;
205 //
206     if(!CryptEccGetParameter(&n, 'n', curveID))
```

```
207             return FALSE;
208
209         // If this is the commit phase, use the current value of the commit counter
210         if(c != NULL)
211         {
212             // if the array bit is not set, can't use the value.
213             if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
214                 return FALSE;
215
216             // If it is the sign phase, figure out what the counter value was
217             // when the commitment was made.
218             //
219             // When gr.commitArray has less than 64K bits, the extra
220             // bits of 'c' are used as a check to make sure that the
221             // signing operation is not using an out of range count value
222             t1 = (UINT16)currentCount;
223
224             // If the lower bits of c are greater or equal to the lower bits of t1
225             // then the upper bits of t1 must be one more than the upper bits
226             // of c
227             if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
228                 // Since the counter is behind, reduce the current count
229                 currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
230
231             t1 = (UINT16)currentCount;
232             if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
233                 return FALSE;
234             // set the counter to the value that was
235             // present when the commitment was made
236             currentCount = (currentCount & 0xffffffffffff0000) | *c;
237         }
238         // Marshal the count value to a TPM2B buffer for the KDF
239         cntr.t.size = sizeof(currentCount);
240         UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
241
242         // Now can do the KDF to create the random value for the signing operation
243         // During the creation process, we may generate an r that does not meet the
244         // requirements of the random value.
245         // want to generate a new r.
246         r->t.size = n.t.size;
247
248         for(iterations = 1; iterations < 1000000;)
249         {
250             int     i;
251             CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
252                       &name->b, &cntr.b, n.t.size * 8, r->t.buffer, &iterations, FALSE);
253
254             // "random" value must be less than the prime
255             if(UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
256                 continue;
257
258             // in this implementation it is required that at least bit
259             // in the upper half of the number be set
260             for(i = n.t.size / 2; i >= 0; i--)
261                 if(r->b.buffer[i] != 0)
262                     return TRUE;
263         }
264         return FALSE;
265     }
```

10.2.11.2.13 CryptCommit()

This function is called when the count value is committed. The *gr.commitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```
266 UINT16
267 CryptCommit(
268     void
269     )
270 {
271     UINT16      oldCount = (UINT16)gr.commitCounter;
272     gr.commitCounter++;
273     SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
274     return oldCount;
275 }
```

10.2.11.2.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.c*ommitArray* bit associated with the count value so that it can't be used again.

```
276 void
277 CryptEndCommit(
278     UINT16          c               // IN: the counter value of the commitment
279     )
280 {
281     ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
282 }
```

10.2.11.2.15 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	unsupported ECC curve ID

```
283 BOOL
284 CryptEccGetParameters(
285     TPM_ECC_CURVE                curveId,       // IN: ECC curve ID
286     TPMS_ALGORITHM_DETAIL_ECC   *parameters     // OUT: ECC parameters
287     )
288 {
289     const ECC_CURVE             *curve = CryptEccGetParametersByCurveId(curveId);
290     const ECC_CURVE_DATA        *data;
291     BOOL                         found = curve != NULL;
292
293     if(found)
294     {
295         data = curve->curveData;
296         parameters->curveID = curve->curveId;
297         parameters->keySize = curve->keySizeBits;
298         parameters->kdf = curve->kdf;
299         parameters->sign = curve->sign;
300 //        BnTo2B(data->prime, &parameters->p.b, 0);
301         BnTo2B(data->prime, &parameters->p.b, parameters->p.t.size);
302         BnTo2B(data->a, &parameters->a.b, 0);
303         BnTo2B(data->b, &parameters->b.b, 0);
304         BnTo2B(data->base.x, &parameters->gX.b, parameters->p.t.size);
```

```
305             BnTo2B(data->base.y, &parameters->gY.b, parameters->p.t.size);
306     //        BnTo2B(data->base.x, &parameters->gX.b, 0);
307     //        BnTo2B(data->base.y, &parameters->gY.b, 0);
308             BnTo2B(data->order, &parameters->n.b, 0);
309             BnTo2B(data->h, &parameters->h.b, 0);
310         }
311         return found;
312     }
```

10.2.11.2.16 BnGetCurvePrime()

This function is used to get just the prime modulus associated with a curve.

```
313     const bignum_t *
314     BnGetCurvePrime(
315         TPM_ECC_CURVE            curveId
316         )
317     {
318         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
319         return (C != NULL) ? CurveGetPrime(C) : NULL;
320     }
```

10.2.11.2.17 BnGetCurveOrder()

This function is used to get just the curve order

```
321     const bignum_t *
322     BnGetCurveOrder(
323         TPM_ECC_CURVE            curveId
324         )
325     {
326         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
327         return (C != NULL) ? CurveGetOrder(C) : NULL;
328     }
```

10.2.11.2.18 BnIsOnCurve()

This function checks if a point is on the curve.

```
329     BOOL
330     BnIsOnCurve(
331         pointConst                   Q,
332         const ECC_CURVE_DATA        *C
333         )
334     {
335         BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
336         BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
337         bigConst                    prime = CurveGetPrime(C);
338     //
339         // Show that point is on the curve y^2 = x^3 + ax + b;
340         // Or y^2 = x(x^2 + a) + b
341         // y^2
342         BnMult(left, Q->y, Q->y);
343
344         BnMod(left, prime);
345     // x^2
346         BnMult(right, Q->x, Q->x);
347
348         // x^2 + a
349         BnAdd(right, right, CurveGet_a(C));
350
```

```
351     //    BnMod(right, CurveGetPrime(C));
352         // x(x^2 + a)
353         BnMult(right, right, Q->x);
354
355         // x(x^2 + a) + b
356         BnAdd(right, right, CurveGet_b(C));
357
358         BnMod(right, prime);
359         if(BnUnsignedCmp(left, right) == 0)
360             return TRUE;
361         else
362             return FALSE;
363     }
```

10.2.11.2.19 BnIsValidPrivateEcc()

Checks that 0 < *x* < *q*

```
364     BOOL
365     BnIsValidPrivateEcc(
366         bigConst                x,          // IN: private key to check
367         bigCurve                E           // IN: the curve to check
368         )
369     {
370         BOOL        retVal;
371         retVal = (!BnEqualZero(x)
372                   && (BnUnsignedCmp(x, CurveGetOrder(AccessCurveData(E))) < 0));
373         return retVal;
374     }
375     LIB_EXPORT BOOL
376     CryptEccIsValidPrivateKey(
377         TPM2B_ECC_PARAMETER     *d,
378         TPM_ECC_CURVE            curveId
379         )
380     {
381         BN_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
382         return !BnEqualZero(bnD) && (BnUnsignedCmp(bnD, BnGetCurveOrder(curveId)) < 0);
383     }
```

10.2.11.2.20 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

```
384     TPM_RC
385     BnPointMult(
386         bigPoint            R,          // OUT: computed point
387         pointConst          S,          // IN: optional point to multiply by 'd'
388         bigConst            d,          // IN: scalar for [d]S or [d]G
389         pointConst          Q,          // IN: optional second point
390         bigConst            u,          // IN: optional second scalar
```

```
391         bigCurve              E           // IN: curve parameters
392         )
393     {
394         BOOL                  OK;
395     //
396         TEST(TPM_ALG_ECDH);
397
398         // Need one scalar
399         OK = (d != NULL || u != NULL);
400
401         // If S is present, then d has to be present. If S is not
402         // present, then d may or may not be present
403         OK = OK && (((S == NULL) == (d == NULL)) || (d != NULL));
404
405         // either both u and Q have to be provided or neither can be provided (don't
406         // know what to do if only one is provided.
407         OK = OK && ((u == NULL) == (Q == NULL));
408
409         OK = OK && (E != NULL);
410         if(!OK)
411             return TPM_RC_VALUE;
412
413         OK = (S == NULL) || BnIsOnCurve(S, AccessCurveData(E));
414         OK = OK && ((Q == NULL) || BnIsOnCurve(Q, AccessCurveData(E)));
415         if(!OK)
416             return TPM_RC_ECC_POINT;
417
418         if((d != NULL) && (S == NULL))
419             S = CurveGetG(AccessCurveData(E));
420         // If only one scalar, don't need Shamir's trick
421         if((d == NULL) || (u == NULL))
422         {
423             if(d == NULL)
424                 OK = BnEccModMult(R, Q, u, E);
425             else
426                 OK = BnEccModMult(R, S, d, E);
427         }
428         else
429         {
430             OK = BnEccModMult2(R, S, d, Q, u, E);
431         }
432         return  (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);
433     }
```

10.2.11.2.21 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

```
434     BOOL
435     BnEccGetPrivate(
436         bigNum                 dOut,       // OUT: the qualified random value
437         const ECC_CURVE_DATA   *C,         // IN: curve for which the private key
438                                //     needs to be appropriate
439         RAND_STATE             *rand       // IN: state for DRBG
440         )
441     {
```

```
442         bigConst                order = CurveGetOrder(C);
443         BOOL                    OK;
444         UINT32                  orderBits = BnSizeInBits(order);
445         UINT32                  orderBytes = BITS_TO_BYTES(orderBits);
446         BN_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
447         BN_VAR(nMinus1, MAX_ECC_KEY_BITS);
448     //
449         OK = BnGetRandomBits(bnExtraBits, (orderBytes * 8) + 64, rand);
450         OK = OK && BnSubWord(nMinus1, order, 1);
451         OK = OK && BnMod(bnExtraBits, nMinus1);
452         OK = OK && BnAddWord(dOut, bnExtraBits, 1);
453         return OK && !g_inFailureMode;
454     }
```

10.2.11.2.22 BnEccGenerateKeyPair()

This function gets a private scalar from the source of random bits and does the point multiply to get the public key.

```
455     BOOL
456     BnEccGenerateKeyPair(
457         bigNum               bnD,            // OUT: private scalar
458         bn_point_t          *ecQ,            // OUT: public point
459         bigCurve             E,              // IN: curve for the point
460         RAND_STATE          *rand            // IN: DRBG state to use
461         )
462     {
463         BOOL                 OK = FALSE;
464         // Get a private scalar
465         OK = BnEccGetPrivate(bnD, AccessCurveData(E), rand);
466
467         // Do a point multiply
468         OK = OK && BnEccModMult(ecQ, NULL, bnD, E);
469         if(!OK)
470             BnSetWord(ecQ->z, 0);
471         else
472             BnSetWord(ecQ->z, 1);
473         return OK;
474     }
```

10.2.11.2.23 CryptEccNewKeyPair

This function creates an ephemeral ECC. It is ephemeral in that is expected that the private part of the key will be discarded

```
475     LIB_EXPORT TPM_RC
476     CryptEccNewKeyPair(
477         TPMS_ECC_POINT          *Qout,      // OUT: the public point
478         TPM2B_ECC_PARAMETER     *dOut,      // OUT: the private scalar
479         TPM_ECC_CURVE            curveId    // IN: the curve for the key
480         )
481     {
482         CURVE_INITIALIZED(E, curveId);
483         POINT(ecQ);
484         ECC_NUM(bnD);
485         BOOL                    OK;
486
487         if(E == NULL)
488             return TPM_RC_CURVE;
489
490         TEST(TPM_ALG_ECDH);
491         OK = BnEccGenerateKeyPair(bnD, ecQ, E, NULL);
```

```
492         if(OK)
493         {
494             BnPointTo2B(Qout, ecQ, E);
495             BnTo2B(bnD, &dOut->b, Qout->x.t.size);
496         }
497         else
498         {
499             Qout->x.t.size = Qout->y.t.size = dOut->t.size = 0;
500         }
501         CURVE_FREE(E);
502         return OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
503     }
```

10.2.11.2.24 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

```
504     LIB_EXPORT TPM_RC
505     CryptEccPointMultiply(
506         TPMS_ECC_POINT      *Rout,              // OUT: the product point R
507         TPM_ECC_CURVE        curveId,           // IN: the curve to use
508         TPMS_ECC_POINT      *Pin,               // IN: first point (can be null)
509         TPM2B_ECC_PARAMETER *dIn,               // IN: scalar value for [dIn]Qin
510                                                 //     the Pin
511         TPMS_ECC_POINT      *Qin,               // IN: point Q
512         TPM2B_ECC_PARAMETER *uIn                // IN: scalar value for the multiplier
513                                                 //     of Q
514         )
515     {
516         CURVE_INITIALIZED(E, curveId);
517         POINT_INITIALIZED(ecP, Pin);
518         ECC_INITIALIZED(bnD, dIn);      // If dIn is null, then bnD is null
519         ECC_INITIALIZED(bnU, uIn);
520         POINT_INITIALIZED(ecQ, Qin);
521         POINT(ecR);
522         TPM_RC              retVal;
523     //
524         retVal = BnPointMult(ecR, ecP, bnD, ecQ, bnU, E);
525
526         if(retVal == TPM_RC_SUCCESS)
527             BnPointTo2B(Rout, ecR, E);
528         else
529             ClearPoint2B(Rout);
530         CURVE_FREE(E);
```

```
531         return retVal;
532     }
```

10.2.11.2.25 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

```
533     LIB_EXPORT BOOL
534     CryptEccIsPointOnCurve(
535         TPM_ECC_CURVE            curveId,       // IN: the curve selector
536         TPMS_ECC_POINT          *Qin            // IN: the point.
537         )
538     {
539         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
540         POINT_INITIALIZED(ecQ, Qin);
541         BOOL             OK;
542     //
543         pAssert(Qin != NULL);
544         OK = (C != NULL && (BnIsOnCurve(ecQ, C)));
545         return OK;
546     }
```

10.2.11.2.26 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)

```
547     LIB_EXPORT TPM_RC
548     CryptEccGenerateKey(
549         TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
550                                                 //      the new key. The public key
551                                                 //      area will be replaced computed
552                                                 //      ECC public key
553         TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
554                                                 //      updated to contain the private
555                                                 //      ECC key and the symmetric
556                                                 //      encryption key
557         RAND_STATE          *rand               // IN: if not NULL, the deterministic
558                                                 //     RNG state
559         )
560     {
561         CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
```

```
562         ECC_NUM(bnD);
563         POINT(ecQ);
564         BOOL                    OK;
565         TPM_RC                  retVal;
566     //
567         TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
568
569         // Validate parameters
570         if(E == NULL)
571             ERROR_RETURN(TPM_RC_CURVE);
572
573         publicArea->unique.ecc.x.t.size = 0;
574         publicArea->unique.ecc.y.t.size = 0;
575         sensitive->sensitive.ecc.t.size = 0;
576
577         OK = BnEccGenerateKeyPair(bnD, ecQ, E, rand);
578         if(OK)
579         {
580             BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
581             BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
582         }
583     #if FIPS_COMPLIANT
584         // See if PWCT is required
585         if(OK && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
586         {
587             ECC_NUM(bnT);
588             ECC_NUM(bnS);
589             TPM2B_DIGEST            digest;
590     //
591             TEST(TPM_ALG_ECDSA);
592             digest.t.size = MIN(sensitive->sensitive.ecc.t.size, sizeof(digest.t.buffer));
593             // Get a random value to sign using the built in DRBG state
594             DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
595             if(g_inFailureMode)
596                 return TPM_RC_FAILURE;
597             BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
598             // and make sure that we can validate the signature
599             OK = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest) == TPM_RC_SUCCESS;
600         }
601     #endif
602         retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
603     Exit:
604         CURVE_FREE(E);
605         return retVal;
606     }
607     #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: [$s = k + r * d \pmod{n}$] 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 $r \pmod{n}$ 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.

```
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 names used in ECSCHNORR making for a lot of confusion.

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 too large. The size of *number* is set so that it has no more bytes of significance than *reference* value. If the resulting number can have more bits of significance than *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* = **Hash**(*xR* || *P*)(mod *q*)

c) *rT* = truncated *r*

d) *s*= *k* + *rT* * *ds* (mod *q*)

e) return the tuple *rT*, *s*

Error Returns	Meaning
TPM_RC_NO_RESULT	failure in the Schnorr sign process
TPM_RC_SCHEME	*hashAlg* can't produce zero-length digest

```
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
```

```
276         const char      *c          //IN: character string number
277         )
278     {
279         ECC_NUM(bnC);
280         BnFromHex(bnC, c);
281         return (BnUnsignedCmp(bn, bnC) == 0);
282     }
283     #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*). This function signs *e*.

Error Returns	Meaning
TPM_RC_VALUE	bad curve

```
284     static TPM_RC
285     BnSignEcSm2(
286         bigNum                  bnR,        // OUT: 'r' component of the signature
287         bigNum                  bnS,        // OUT: 's' component of the signature
288         bigCurve                E,          // IN: the curve used in signing
289         bigNum                  bnD,        // IN: the private key
290         const TPM2B_DIGEST      *digest,    // IN: the digest to sign
291         RAND_STATE              *rand       // IN: random number generator (mostly for
292                                             //     debug)
293         )
294     {
295         BN_MAX_INITIALIZED(bnE, digest);    // Don't know how big digest might be
296         ECC_NUM(bnN);
297         ECC_NUM(bnK);
298         ECC_NUM(bnT);                       // temp
299         POINT(Q1);
300         bigConst                    order = (E != NULL)
301             ? CurveGetOrder(AccessCurveData(E)) : NULL;
302     //
303     #ifdef _SM2_SIGN_DEBUG
304         BnFromHex(bnE, "B524F552CD82B8B028476E005C377FB1"
305                        "9A87E6FC682D48BB5D42E3D9B9EFFE76");
306         BnFromHex(bnD, "128B2FA8BD433C6C068C8D803DFF7979"
307                        "2A519A55171B1B650C23661D15897263");
308     #endif
309         // A3: Use random number generator to generate random number 1 <= k <= n-1;
310         // NOTE: Ax: numbers are from the SM2 standard
311     loop:
312         {
313             // Get a random number 0 < k < n
314             BnGenerateRandomInRange(bnK, order, rand);
315     #ifdef _SM2_SIGN_DEBUG
316             BnFromHex(bnK, "6CB28D99385C175C94F94E934817663F"
317                            "C176D925DD72B727260DBAAE1FB2F96F");
318     #endif
319             // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
320             // to details specified in 4.2.7 in Part 1 of this document, transform the
321             // data type of x1 into an integer;
322             if(!BnEccModMult(Q1, NULL, bnK, E))
323                 goto loop;
324             // A5: Figure out 'r' = ('e' + 'x1') mod 'n',
325             BnAdd(bnR, bnE, Q1->x);
326             BnMod(bnR, order);
327     #ifdef _SM2_SIGN_DEBUG
328             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

```
#define _CRYPT_HASH_C_
#include "Tpm.h"
#include "CryptHash_fp.h"
#include "CryptHash.h"
#include "OIDs.h"
#define HASH_TABLE_SIZE     (HASH_COUNT + 1)
#if     ALG_SHA1
HASH_DEF_TEMPLATE(SHA1, Sha1);
#endif
#if     ALG_SHA256
HASH_DEF_TEMPLATE(SHA256, Sha256);
#endif
#if     ALG_SHA384
HASH_DEF_TEMPLATE(SHA384, Sha384);
#endif
#if     ALG_SHA512
HASH_DEF_TEMPLATE(SHA512, Sha512);
#endif
#if ALG_SM3_256
HASH_DEF_TEMPLATE(SM3_256, Sm3_256);
#endif
HASH_DEF NULL_Def = {{0}};

PHASH_DEF       HashDefArray[] = {
#if ALG_SHA1
    &Sha1_Def,
#endif
#if ALG_SHA256
    &Sha256_Def,
#endif
#if ALG_SHA384
    &Sha384_Def,
#endif
#if ALG_SHA512
    &Sha512_Def,
#endif
#if ALG_SM3_256
    &Sm3_256_Def,
#endif
    &NULL_Def
};

//** Obligatory Initialization Functions

//*** CryptHashInit()
// This function is called by _TPM_Init do perform the initialization operations for
// the library.
BOOL
CryptHashInit(
    void
    )
{
    LibHashInit();
    return TRUE;
```

```
55  }
```

10.2.13.2.1 CryptHashStartup()

This function is called by TPM2_Startup(). It checks that the size of the HashDefArray() is consistent with the HASH_COUNT.

```
56  BOOL
57  CryptHashStartup(
58      void
59      )
60  {
61      int         i = sizeof(HashDefArray) / sizeof(PHASH_DEF) - 1;
62      return (i == HASH_COUNT);
63  }
```

10.2.13.3 Hash Information Access Functions

10.2.13.3.1 Introduction

These functions provide access to the hash algorithm description information.

10.2.13.3.2 CryptGetHashDef()

This function accesses the hash descriptor associated with a hash a algorithm. The function returns a pointer to a *null* descriptor if *hashAlg* is TPM_ALG_NULL or not a defined algorithm.

```
64  PHASH_DEF
65  CryptGetHashDef(
66      TPM_ALG_ID       hashAlg
67      )
68  {
69      size_t           i;
70  #define HASHES  (sizeof(HashDefArray) / sizeof(PHASH_DEF))
71      for(i = 0; i < HASHES; i++)
72      {
73          PHASH_DEF p = HashDefArray[i];
74          if(p->hashAlg == hashAlg)
75              return p;
76      }
77      return &NULL_Def;
78  }
```

10.2.13.3.3 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

```
79  BOOL
80  CryptHashIsValidAlg(
81      TPM_ALG_ID       hashAlg,           // IN: the algorithm to check
82      BOOL             flag               // IN: TRUE if TPM_ALG_NULL is to be treated
83                                          //     as a valid hash
```

```
84      )
85  {
86      if(hashAlg == TPM_ALG_NULL)
87          return flag;
88      return CryptGetHashDef(hashAlg) != &NULL_Def;
89  }
```

10.2.13.3.4 CryptHashGetAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_xxx	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
90  LIB_EXPORT TPM_ALG_ID
91  CryptHashGetAlgByIndex(
92      UINT32          index           // IN: the index
93      )
94  {
95      TPM_ALG_ID      hashAlg;
96      if(index >= HASH_COUNT)
97          hashAlg = TPM_ALG_NULL;
98      else
99          hashAlg = HashDefArray[index]->hashAlg;
100     return hashAlg;
101 }
```

10.2.13.3.5 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

```
102 LIB_EXPORT UINT16
103 CryptHashGetDigestSize(
104     TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
105     )
106 {
107     return CryptGetHashDef(hashAlg)->digestSize;
108 }
```

10.2.13.3.6 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

```
109   LIB_EXPORT UINT16
110   CryptHashGetBlockSize(
111       TPM_ALG_ID       hashAlg        // IN: hash algorithm to look up
112       )
113   {
114       return CryptGetHashDef(hashAlg)->blockSize;
115   }
```

10.2.13.3.7 CryptHashGetOid()

This function returns a pointer to DER=encoded OID for a hash algorithm. All OIDs are full OID values including the Tag (0x06) and length byte.

```
116   LIB_EXPORT const BYTE *
117   CryptHashGetOid(
118       TPM_ALG_ID      hashAlg
119   )
120   {
121       return CryptGetHashDef(hashAlg)->OID;
122   }
```

10.2.13.3.8 CryptHashGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
123   TPM_ALG_ID
124   CryptHashGetContextAlg(
125       PHASH_STATE      state          // IN: the context to check
126       )
127   {
128       return state->hashAlg;
129   }
```

10.2.13.4 State Import and Export

10.2.13.4.1 CryptHashCopyState

This function is used to clone a HASH_STATE.

```
130   LIB_EXPORT void
131   CryptHashCopyState(
132       HASH_STATE          *out,          // OUT: destination of the state
133       const HASH_STATE    *in            // IN: source of the state
134       )
135   {
136       pAssert(out->type == in->type);
137       out->hashAlg = in->hashAlg;
138       out->def = in->def;
139       if(in->hashAlg != TPM_ALG_NULL)
140       {
141           HASH_STATE_COPY(out, in);
142       }
143       if(in->type == HASH_STATE_HMAC)
144       {
```

```
145             const HMAC_STATE    *hIn = (HMAC_STATE *)in;
146             HMAC_STATE          *hOut = (HMAC_STATE *)out;
147             hOut->hmacKey = hIn->hmacKey;
148         }
149         return;
150     }
```

10.2.13.4.2 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.

```
151     void
152     CryptHashExportState(
153         PCHASH_STATE         internalFmt,   // IN: the hash state formatted for use by
154                                             //     library
155         PEXPORT_HASH_STATE   externalFmt    // OUT: the exported hash state
156         )
157     {
158         BYTE                    *outBuf = (BYTE *)externalFmt;
159     //
160         cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
161         // the following #define is used to move data from an aligned internal data
162         // structure to a byte buffer (external format data.
163     #define CopyToOffset(value)                                                 \
164             memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value,    \
165                     sizeof(internalFmt->value))
166         // Copy the hashAlg
167         CopyToOffset(hashAlg);
168         CopyToOffset(type);
169     #ifdef HASH_STATE_SMAC
170         if(internalFmt->type == HASH_STATE_SMAC)
171         {
172             memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
173             return;
174
175         }
176     #endif
177         if(internalFmt->type == HASH_STATE_HMAC)
178         {
179             HMAC_STATE              *from = (HMAC_STATE *)internalFmt;
180             memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)], &from->hmacKey,
181                    sizeof(from->hmacKey));
182         }
183         if(internalFmt->hashAlg != TPM_ALG_NULL)
184              HASH_STATE_EXPORT(externalFmt, internalFmt);
185     }
```

10.2.13.4.3 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.

```
186     void
187     CryptHashImportState(
188         PHASH_STATE          internalFmt,   // OUT: the hash state formatted for use by
189                                             //     the library
190         PCEXPORT_HASH_STATE  externalFmt    // IN: the exported hash state
191         )
192     {
193         BYTE                    *inBuf = (BYTE *)externalFmt;
194     //
```

```
195 #define CopyFromOffset(value)                                                \
196     memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)],          \
197            sizeof(internalFmt->value))
198
199     // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
200     // internal structure.
201     CopyFromOffset(hashAlg);
202     CopyFromOffset(type);
203     if(internalFmt->hashAlg != TPM_ALG_NULL)
204     {
205 #ifdef HASH_STATE_SMAC
206         if(internalFmt->type == HASH_STATE_SMAC)
207         {
208             memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
209             return;
210         }
211 #endif
212         internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
213         HASH_STATE_IMPORT(internalFmt, inBuf);
214         if(internalFmt->type == HASH_STATE_HMAC)
215         {
216             HMAC_STATE              *to = (HMAC_STATE *)internalFmt;
217             memcpy(&to->hmacKey, &inBuf[offsetof(HMAC_STATE, hmacKey)],
218                    sizeof(to->hmacKey));
219         }
220     }
221 }
```

10.2.13.5 State Modification Functions

10.2.13.5.1 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.

```
222 static UINT16
223 HashEnd(
224     PHASH_STATE      hashState,     // IN: the hash state
225     UINT32           dOutSize,      // IN: the size of receive buffer
226     PBYTE            dOut           // OUT: the receive buffer
227     )
228 {
229     BYTE                temp[MAX_DIGEST_SIZE];
230     if((hashState->hashAlg == TPM_ALG_NULL)
231        || (hashState->type != HASH_STATE_HASH))
232         dOutSize = 0;
233     if(dOutSize > 0)
234     {
235         hashState->def = CryptGetHashDef(hashState->hashAlg);
236         // Set the final size
237         dOutSize = MIN(dOutSize, hashState->def->digestSize);
238         // Complete into the temp buffer and then copy
239         HASH_END(hashState, temp);
240         // Don't want any other functions calling the HASH_END method
241         // directly.
242 #undef HASH_END
243         memcpy(dOut, &temp, dOutSize);
244     }
245     hashState->type = HASH_STATE_EMPTY;
246     return (UINT16)dOutSize;
247 }
```

10.2.13.5.2 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

```
248  LIB_EXPORT UINT16
249  CryptHashStart(
250      PHASH_STATE      hashState,      // OUT: the running hash state
251      TPM_ALG_ID       hashAlg         // IN: hash algorithm
252      )
253  {
254      UINT16               retVal;
255
256      TEST(hashAlg);
257
258      hashState->hashAlg = hashAlg;
259      if(hashAlg == TPM_ALG_NULL)
260      {
261          retVal = 0;
262      }
263      else
264      {
265          hashState->def = CryptGetHashDef(hashAlg);
266          HASH_START(hashState);
267          retVal = hashState->def->digestSize;
268      }
269  #undef HASH_START
270      hashState->type = HASH_STATE_HASH;
271      return retVal;
272  }
```

10.2.13.5.3 CryptDigestUpdate()

Add data to a hash or HMAC, SMAC stack.

```
273  void
274  CryptDigestUpdate(
275      PHASH_STATE      hashState,      // IN: the hash context information
276      UINT32           dataSize,       // IN: the size of data to be added
277      const BYTE      *data            // IN: data to be hashed
278      )
279  {
280      if(hashState->hashAlg != TPM_ALG_NULL)
281      {
282          if((hashState->type == HASH_STATE_HASH)
283             || (hashState->type == HASH_STATE_HMAC))
284              HASH_DATA(hashState, dataSize, (BYTE *)data);
285  #if SMAC_IMPLEMENTED
286          else if(hashState->type == HASH_STATE_SMAC)
287              (hashState->state.smac.smacMethods.data)(&hashState->state.smac.state,
288                                                      dataSize, data);
```

```
289     #endif // SMAC_IMPLEMENTED
290             else
291                 FAIL(FATAL_ERROR_INTERNAL);
292         }
293         return;
294     }
```

10.2.13.5.4 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

```
295     LIB_EXPORT UINT16
296     CryptHashEnd(
297         PHASH_STATE      hashState,     // IN: the state of hash stack
298         UINT32           dOutSize,      // IN: size of digest buffer
299         BYTE            *dOut           // OUT: hash digest
300         )
301     {
302         pAssert(hashState->type == HASH_STATE_HASH);
303         return HashEnd(hashState, dOutSize, dOut);
304     }
```

10.2.13.5.5 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*

```
305     LIB_EXPORT UINT16
306     CryptHashBlock(
307         TPM_ALG_ID       hashAlg,       // IN: The hash algorithm
308         UINT32           dataSize,      // IN: size of buffer to hash
309         const BYTE      *data,          // IN: the buffer to hash
310         UINT32           dOutSize,      // IN: size of the digest buffer
311         BYTE            *dOut           // OUT: digest buffer
312         )
313     {
314         HASH_STATE          state;
315         CryptHashStart(&state, hashAlg);
316         CryptDigestUpdate(&state, dataSize, data);
317         return HashEnd(&state, dOutSize, dOut);
318     }
```

10.2.13.5.6 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
    )
{
    // Only compute the digest if a pointer to the 2B is provided.
    // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
    // to the digest occurs. This function should not provide a buffer if bIn is
    // not provided.
    pAssert(bIn != NULL);
    CryptDigestUpdate(state, bIn->size, bIn->buffer);
    return;
}
```

10.2.13.5.7 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*

```
LIB_EXPORT UINT16
CryptHashEnd2B(
    PHASH_STATE      state,          // IN: the hash state
    P2B              digest          // IN: the size of the buffer Out: requested
                                     //     number of bytes
    )
{
    return CryptHashEnd(state, digest->size, digest->buffer);
}
```

10.2.13.5.8 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().

```
LIB_EXPORT void
CryptDigestUpdateInt(
    void            *state,          // IN: the state of hash stack
    UINT32           intSize,        // IN: the size of 'intValue' in bytes
    UINT64           intValue        // IN: integer value to be hashed
    )
{
#if LITTLE_ENDIAN_TPM
    intValue = REVERSE_ENDIAN_64(intValue);
#endif
    CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
}
```


10.2.13.6 HMAC Functions

10.2.13.6.1 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)

```
354  LIB_EXPORT UINT16
355  CryptHmacStart(
356      PHMAC_STATE      state,          // IN/OUT: the state buffer
357      TPM_ALG_ID       hashAlg,        // IN: the algorithm to use
358      UINT16           keySize,        // IN: the size of the HMAC key
359      const BYTE      *key             // IN: the HMAC key
360      )
361  {
362      PHASH_DEF               hashDef;
363      BYTE *                  pb;
364      UINT32                  i;
365  //
366      hashDef = CryptGetHashDef(hashAlg);
367      if(hashDef->digestSize != 0)
368      {
369      // If the HMAC key is larger than the hash block size, it has to be reduced
370      // to fit. The reduction is a digest of the hashKey.
371          if(keySize > hashDef->blockSize)
372          {
373              // if the key is too big, reduce it to a digest of itself
374              state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
375                                                     hashDef->digestSize,
376                                                     state->hmacKey.t.buffer);
377          }
378          else
379          {
380              memcpy(state->hmacKey.t.buffer, key, keySize);
381              state->hmacKey.t.size = keySize;
382          }
383          // XOR the key with iPad (0x36)
384          pb = state->hmacKey.t.buffer;
385          for(i = state->hmacKey.t.size; i > 0; i--)
386              *pb++ ^= 0x36;
387
388          // if the keySize is smaller than a block, fill the rest with 0x36
389          for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
390              *pb++ = 0x36;
391
392          // Increase the oPadSize to a full block
393          state->hmacKey.t.size = hashDef->blockSize;
394
395          // Start a new hash with the HMAC key
396          // This will go in the caller's state structure and may be a sequence or not
397          CryptHashStart((PHASH_STATE)state, hashAlg);
398          CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
399                            state->hmacKey.t.buffer);
400          // XOR the key block with 0x5c ^ 0x36
401          for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
402              *pb++ ^= (0x5c ^ 0x36);
403      }
404      // Set the hash algorithm
```

```
405         state->hashState.hashAlg = hashAlg;
406         // Set the hash state type
407         state->hashState.type = HASH_STATE_HMAC;
408 
409         return hashDef->digestSize;
410     }
```

10.2.13.6.2 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)

```
411     LIB_EXPORT UINT16
412     CryptHmacEnd(
413         PHMAC_STATE      state,          // IN: the hash state buffer
414         UINT32           dOutSize,       // IN: size of digest buffer
415         BYTE            *dOut            // OUT: hash digest
416         )
417     {
418         BYTE                 temp[MAX_DIGEST_SIZE];
419         PHASH_STATE          hState = (PHASH_STATE)&state->hashState;
420 
421     #if SMAC_IMPLEMENTED
422         if(hState->type == HASH_STATE_SMAC)
423             return (state->hashState.state.smac.smacMethods.end)
424                         (&state->hashState.state.smac.state,
425                          dOutSize,
426                          dOut);
427     #endif
428         pAssert(hState->type == HASH_STATE_HMAC);
429         hState->def = CryptGetHashDef(hState->hashAlg);
430         // Change the state type for completion processing
431         hState->type = HASH_STATE_HASH;
432         if(hState->hashAlg == TPM_ALG_NULL)
433             dOutSize = 0;
434         else
435         {
436 
437         // Complete the current hash
438             HashEnd(hState, hState->def->digestSize, temp);
439             // Do another hash starting with the oPad
440             CryptHashStart(hState, hState->hashAlg);
441             CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
442             CryptDigestUpdate(hState, hState->def->digestSize, temp);
443         }
444         return HashEnd(hState, dOutSize, dOut);
445     }
```

10.2.13.6.3 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

```
446 LIB_EXPORT UINT16
447 CryptHmacStart2B(
448     PHMAC_STATE      hmacState,     // OUT: the state of HMAC stack. It will be used
449                                     //      in HMAC update and completion
450     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
451     P2B              key            // IN: HMAC key
452     )
453 {
454     return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
455 }
```

10.2.13.6.4 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*

```
456 LIB_EXPORT UINT16
457 CryptHmacEnd2B(
458     PHMAC_STATE      hmacState,     // IN: the state of HMAC stack
459     P2B              digest         // OUT: HMAC
460     )
461 {
462     return CryptHmacEnd(hmacState, digest->size, digest->buffer);
463 }
```

10.2.13.7 Mask and Key Generation Functions

10.2.13.7.1 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*

```
464 LIB_EXPORT UINT16
465 CryptMGF1(
466     UINT32           mSize,         // IN: length of the mask to be produced
467     BYTE            *mask,          // OUT: buffer to receive the mask
468     TPM_ALG_ID       hashAlg,       // IN: hash to use
469     UINT32           seedSize,      // IN: size of the seed
470     BYTE            *seed           // IN: seed size
471     )
472 {
473     HASH_STATE           hashState;
474     PHASH_DEF            hDef = CryptGetHashDef(hashAlg);
475     UINT32               remaining;
```

```
476         UINT32              counter = 0;
477         BYTE                swappedCounter[4];
478
479         // If there is no digest to compute return
480         if((hashAlg == TPM_ALG_NULL) || (mSize == 0))
481             return 0;
482
483         for(remaining = mSize; ; remaining -= hDef->digestSize)
484         {
485             // Because the system may be either Endian...
486             UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
487
488             // Start the hash and include the seed and counter
489             CryptHashStart(&hashState, hashAlg);
490             CryptDigestUpdate(&hashState, seedSize, seed);
491             CryptDigestUpdate(&hashState, 4, swappedCounter);
492
493             // Handling the completion depends on how much space remains in the mask
494             // buffer. If it can hold the entire digest, put it there. If not
495             // put the digest in a temp buffer and only copy the amount that
496             // will fit into the mask buffer.
497             HashEnd(&hashState, remaining, mask);
498             if(remaining <= hDef->digestSize)
499                 break;
500             mask = &mask[hDef->digestSize];
501             counter++;
502         }
503         return (UINT16)mSize;
504     }
```

10.2.13.7.2 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

```
505     LIB_EXPORT UINT16
506     CryptKDFa(
507         TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
508         const TPM2B     *key,           // IN: HMAC key
509         const TPM2B     *label,         // IN: a label for the KDF
510         const TPM2B     *contextU,      // IN: context U
511         const TPM2B     *contextV,      // IN: context V
512         UINT32           sizeInBits,    // IN: size of generated key in bits
513         BYTE            *keyStream,     // OUT: key buffer
514         UINT32          *counterInOut,  // IN/OUT: caller may provide the iteration
515                                         //     counter for incremental operations to
```

```
                                        //     avoid large intermediate buffers.
    UINT16          blocks              // IN: If non-zero, this is the maximum number
                                        //     of blocks to be returned, regardless
                                        //     of sizeInBits
    )
{
    UINT32                  counter = 0;        // counter value
    INT16                   bytes;              // number of bytes to produce
    UINT16                  generated;          // number of bytes generated
    BYTE                   *stream = keyStream;
    HMAC_STATE              hState;
    UINT16                  digestSize = CryptHashGetDigestSize(hashAlg);

    pAssert(key != NULL && keyStream != NULL);

    TEST(TPM_ALG_KDF1_SP800_108);

    if(digestSize == 0)
        return 0;

    if(counterInOut != NULL)
        counter = *counterInOut;

    // If the size of the request is larger than the numbers will handle,
    // it is a fatal error.
    pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);

    // The number of bytes to be generated is the smaller of the sizeInBits bytes or
    // the number of requested blocks. The number of blocks is the smaller of the
    // number requested or the number allowed by sizeInBits. A partial block is
    // a full block.
    bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
    generated = bytes;

    // Generate required bytes
    for(; bytes > 0; bytes -= digestSize)
    {
        counter++;
        // Start HMAC
        if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
            return 0;
        // Adding counter
        CryptDigestUpdateInt(&hState.hashState, 4, counter);

        // Adding label
        if(label != NULL)
            HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
        // Add a null. SP108 is not very clear about when the 0 is needed but to
        // make this like the previous version that did not add an 0x00 after
        // a null-terminated string, this version will only add a null byte
        // if the label parameter did not end in a null byte, or if no label
        // is present.
        if((label == NULL)
           || (label->size == 0)
           || (label->buffer[label->size - 1] != 0))
            CryptDigestUpdateInt(&hState.hashState, 1, 0);
        // Adding contextU
        if(contextU != NULL)
            HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
        // Adding contextV
        if(contextV != NULL)
            HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
        // Adding size in bits
        CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);

        // Complete and put the data in the buffer
```

```
582             CryptHmacEnd(&hState, bytes, stream);
583             stream = &stream[digestSize];
584         }
585         // Masking in the KDF is disabled. If the calling function wants something
586         // less than even number of bytes, then the caller should do the masking
587         // because there is no universal way to do it here
588         if(counterInOut != NULL)
589             *counterInOut = counter;
590         return generated;
591     }
```

10.2.13.7.3 CryptKDFe()

This function implements 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

```
592     LIB_EXPORT UINT16
593     CryptKDFe(
594         TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
595         TPM2B           *Z,             // IN: Z
596         const TPM2B     *label,         // IN: a label value for the KDF
597         TPM2B           *partyUInfo,    // IN: PartyUInfo
598         TPM2B           *partyVInfo,    // IN: PartyVInfo
599         UINT32           sizeInBits,    // IN: size of generated key in bits
600         BYTE            *keyStream      // OUT: key buffer
601         )
602     {
603         HASH_STATE       hashState;
604         PHASH_DEF        hashDef = CryptGetHashDef(hashAlg);
605
606         UINT32           counter = 0;       // counter value
607         UINT16           hLen;
608         BYTE            *stream = keyStream;
609         INT16            bytes;             // number of bytes to generate
610
611         pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
612     //
613         hLen = hashDef->digestSize;
614         bytes = (INT16)((sizeInBits + 7) / 8);
615         if(hashAlg == TPM_ALG_NULL || bytes == 0)
616             return 0;
617
618         // Generate required bytes
619         //The inner loop of that KDF uses:
620         //  Hash[i] := H(counter | Z | OtherInfo) (5)
621         // Where:
622         //  Hash[i]         the hash generated on the i-th iteration of the loop.
623         //  H()             an approved hash function
624         //  counter         a 32-bit counter that is initialized to 1 and incremented
625         //                  on each iteration
626         //  Z               the X coordinate of the product of a public ECC key and a
627         //                  different private ECC key.
628         //  OtherInfo       a collection of qualifying data for the KDF defined below.
629         //  In this specification, OtherInfo will be constructed by:
```

```
630         //      OtherInfo := Use | PartyUInfo  | PartyVInfo
631         for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
632         {
633             if(bytes < hLen)
634                 hLen = bytes;
635             counter++;
636             // Do the hash
637             CryptHashStart(&hashState, hashAlg);
638             // Add counter
639             CryptDigestUpdateInt(&hashState, 4, counter);
640 
641             // Add Z
642             if(Z != NULL)
643                 CryptDigestUpdate2B(&hashState, Z);
644             // Add label
645             if(label != NULL)
646                 CryptDigestUpdate2B(&hashState, label);
647             // Add a null. SP108 is not very clear about when the 0 is needed but to
648             // make this like the previous version that did not add an 0x00 after
649             // a null-terminated string, this version will only add a null byte
650             // if the label parameter did not end in a null byte, or if no label
651             // is present.
652             if((label == NULL)
653                || (label->size == 0)
654                || (label->buffer[label->size - 1] != 0))
655                 CryptDigestUpdateInt(&hashState, 1, 0);
656             // Add PartyUInfo
657             if(partyUInfo != NULL)
658                 CryptDigestUpdate2B(&hashState, partyUInfo);
659 
660             // Add PartyVInfo
661             if(partyVInfo != NULL)
662                 CryptDigestUpdate2B(&hashState, partyVInfo);
663 
664             // Compute Hash. hLen was changed to be the smaller of bytes or hLen
665             // at the start of each iteration.
666             CryptHashEnd(&hashState, hLen, stream);
667         }
668 
669         // Mask off bits if the required bits is not a multiple of byte size
670         if((sizeInBits % 8) != 0)
671             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
672 
673         return (UINT16)((sizeInBits + 7) / 8);
674     }
```

10.2.14 CryptPrime.c

10.2.14.1 Introduction

This file contains the code for prime validation.

```
#include "Tpm.h"
#include "CryptPrime_fp.h"
//#define CPRI_PRIME
//#include "PrimeTable.h"
#include "CryptPrimeSieve_fp.h"
extern const uint32_t       s_LastPrimeInTable;
extern const uint32_t       s_PrimeTableSize;
extern const uint32_t       s_PrimesInTable;
extern const unsigned char s_PrimeTable[];
extern bigConst             s_CompositeOfSmallPrimes;

//** Functions

//*** Root2()
// This finds ceil(sqrt(n)) to use as a stopping point for searching the prime
// table.
static uint32_t
Root2(
    uint32_t             n
    )
{
    int32_t              last = (int32_t)(n >> 2);
    int32_t              next = (int32_t)(n >> 1);
    int32_t              diff;
    int32_t              stop = 10;
//
    // get a starting point
    for(; next != 0; last >>= 1, next >>= 2);
    last++;
    do
    {
        next = (last + (n / last)) >> 1;
        diff = next - last;
        last = next;
        if(stop-- == 0)
            FAIL(FATAL_ERROR_INTERNAL);
    } while(diff < -1 || diff > 1);
    if((n / next) > (unsigned)next)
        next++;
    pAssert(next != 0);
    pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
    return next;
}
```

10.2.14.1.1 IsPrimeInt()

This will do a test of a word of up to 32-bits in size.

```
BOOL
IsPrimeInt(
    uint32_t            n
    )
{
    uint32_t            i;
    uint32_t            stop;
    if(n < 3 || ((n & 1) == 0))
```

```
52          return (n == 2);
53      if(n <= s_LastPrimeInTable)
54      {
55          n >>= 1;
56          return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
57      }
58      // Need to search
59      stop = Root2(n) >> 1;
60      // starting at 1 is equivalent to staring at  (1 << 1) + 1 = 3
61      for(i = 1; i < stop; i++)
62      {
63          if((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
64              // see if this prime evenly divides the number
65              if((n % ((i << 1) + 1)) == 0)
66                  return FALSE;
67      }
68      return TRUE;
69  }
```

10.2.14.1.2 BnIsProbablyPrime()

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.

```
70  BOOL
71  BnIsProbablyPrime(
72      bigNum          prime,          // IN:
73      RAND_STATE      *rand           // IN: the random state just
74                                      //     in case Miller-Rabin is required
75      )
76  {
77  #if RADIX_BITS > 32
78      if(BnUnsignedCmpWord(prime, UINT32_MAX) <= 0)
79  #else
80      if(BnGetSize(prime) == 1)
81  #endif
82          return IsPrimeInt((uint32_t)prime->d[0]);
83
84      if(BnIsEven(prime))
85          return FALSE;
86      if(BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
87      {
88          crypt_uword_t       temp = prime->d[0] >> 1;
89          return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
90      }
91      {
92          BN_VAR(n, LARGEST_NUMBER_BITS);
93          BnGcd(n, prime, s_CompositeOfSmallPrimes);
94          if(!BnEqualWord(n, 1))
95              return FALSE;
96      }
97      return MillerRabin(prime, rand);
98  }
```

10.2.14.1.3 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.

```
99  UINT32
100 MillerRabinRounds(
101     UINT32          bits            // IN: Number of bits in the RSA prime
```

```
102         )
103     {
104         if(bits < 511) return 8;     // don't really expect this
105         if(bits < 1536) return 5;    // for 512 and 1K primes
106         return 4;                    // for 3K public modulus and greater
107     }
```

10.2.14.1.4 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

```
108     BOOL
109     MillerRabin(
110         bigNum           bnW,
111         RAND_STATE      *rand
112         )
113     {
114         BN_MAX(bnWm1);
115         BN_PRIME(bnM);
116         BN_PRIME(bnB);
117         BN_PRIME(bnZ);
118         BOOL             ret = FALSE;   // Assumed composite for easy exit
119         unsigned int     a;
120         unsigned int     j;
121         int              wLen;
122         int              i;
123         int              iterations = MillerRabinRounds(BnSizeInBits(bnW));
124     //
125        INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
126
127         pAssert(bnW->size > 1);
128         // Let a be the largest integer such that 2^a divides w1.
129         BnSubWord(bnWm1, bnW, 1);
130         pAssert(bnWm1->size != 0);
131
132         // Since w is odd (w-1) is even so start at bit number 1 rather than 0
133         // Get the number of bits in bnWm1 so that it doesn't have to be recomputed
134         // on each iteration.
135         i = (int)(bnWm1->size * RADIX_BITS);
136       // Now find the largest power of 2 that divides w1
137         for(a = 1;
138         (a < (bnWm1->size * RADIX_BITS)) &&
139             (BnTestBit(bnWm1, a) == 0);
140             a++);
141      // 2. m = (w1) / 2^a
142             BnShiftRight(bnM, bnWm1, a);
143           // 3. wlen = len (w).
144         wLen = BnSizeInBits(bnW);
145       // 4. For i = 1 to iterations do
146         for(i = 0; i < iterations; i++)
147         {
148             // 4.1 Obtain a string b of wlen bits from an RBG.
149             // Ensure that 1 < b < w1.
150             // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
151             while(BnGetRandomBits(bnB, wLen, rand) && ((BnUnsignedCmpWord(bnB, 1) <= 0)
152                 || (BnUnsignedCmp(bnB, bnWm1) >= 0)));
153             if(g_inFailureMode)
```

```
154             return FALSE;
155
156         // 4.3 z = b^m mod w.
157         // if ModExp fails, then say this is not
158         // prime and bail out.
159         BnModExp(bnZ, bnB, bnM, bnW);
160
161         // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
162         if((BnUnsignedCmpWord(bnZ, 1) == 0)
163            || (BnUnsignedCmp(bnZ, bnWm1) == 0))
164             goto step4point7;
165         // 4.5 For j = 1 to a  1 do.
166         for(j = 1; j < a; j++)
167         {
168           // 4.5.1 z = z^2 mod w.
169             BnModMult(bnZ, bnZ, bnZ, bnW);
170           // 4.5.2 If (z = w1), then go to step 4.7.
171             if(BnUnsignedCmp(bnZ, bnWm1) == 0)
172                 goto step4point7;
173           // 4.5.3 If (z = 1), then go to step 4.6.
174             if(BnEqualWord(bnZ, 1))
175                 goto step4point6;
176         }
177         // 4.6 Return COMPOSITE.
178 step4point6:
179         INSTRUMENT_INC(failedAtIteration[i]);
180         goto end;
181       // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
182 step4point7:
183         continue;
184     }
185   // 5. Return PROBABLY PRIME
186     ret = TRUE;
187 end:
188     return ret;
189 }
190 #if ALG_RSA
```

10.2.14.1.5 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.

```
191 TPM_RC
192 RsaCheckPrime(
193     bigNum           prime,
194     UINT32           exponent,
195     RAND_STATE      *rand
196     )
197 {
198 #if !RSA_KEY_SIEVE
199     TPM_RC           retVal = TPM_RC_SUCCESS;
200     UINT32           modE = BnModWord(prime, exponent);
201
202     NOT_REFERENCED(rand);
203
204     if(modE == 0)
205         // evenly divisible so add two keeping the number odd
206         BnAddWord(prime, prime, 2);
```

```
207         // want 0 != (p - 1) mod e
208         // which is 1 != p mod e
209         else if(modE == 1)
210             // subtract 2 keeping number odd and insuring that
211             // 0 != (p - 1) mod e
212             BnSubWord(prime, prime, 2);
213
214         if(BnIsProbablyPrime(prime, rand) == 0)
215             ERROR_RETURN(g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_VALUE);
216     Exit:
217         return retVal;
218     #else
219         return PrimeSelectWithSieve(prime, exponent, rand);
220     #endif
221     }
```

10.2.14.1.6 AdjustPrimeCandiate()

For this math, we assume that the RSA numbers are fixed-point numbers with the decimal point to the **left** of the most significant bit. This approach helps make it clear what is happening with the MSb of the values. The two RSA primes have to be large enough so that their product will be a number with the necessary number of significant bits. For example, we want to be able to multiply two 1024-bit numbers to produce a number with 2048 significant bits. If we accept any 1024-bit prime that has its MSb set, then it is possible to produce a product that does not have the MSb SET. For example, if we use tiny keys of 16 bits and have two 8-bit *primes* of 0x80, then the public key would be 0x4000 which is only 15-bits. So, what we need to do is made sure that each of the primes is large enough so that the product of the primes is twice as large as each prime. A little arithmetic will show that the only way to do this is to make sure that each of the primes is no less than root(2)/2. That's what this functions does. 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 (0.7071067811865475) approximated with *0xB505* which is, in fixed point, 0.7071075439453125 or an error of 0.000108%. 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.

This function can be replaced with a function that just sets the two most significant bits of each prime candidate without introducing any computational issues.

```
222     LIB_EXPORT void
223     RsaAdjustPrimeCandidate(
224         bigNum          prime
225         )
226     {
227         UINT32          msw;
228         UINT32          adjusted;
229
230         // If the radix is 32, the compiler should turn this into a simple assignment
231         msw = prime->d[prime->size - 1] >> ((RADIX_BITS == 64) ? 32 : 0);
232         // Multiplying 0xff...f by 0x4AFB gives 0xff..f - 0xB5050...0
233         adjusted = (msw >> 16) * 0x4AFB;
234         adjusted += ((msw & 0xFFFF) * 0x4AFB) >> 16;
235         adjusted += 0xB5050000UL;
236     #if RADIX_BITS == 64
237         // Save the low-order 32 bits
238         prime->d[prime->size - 1] &= 0xFFFFFFFFUL;
239         // replace the upper 32-bits
240         prime->d[prime->size -1] |= ((crypt_uword_t)adjusted << 32);
241     #else
242         prime->d[prime->size - 1] = (crypt_uword_t)adjusted;
243     #endif
244         // make sure the number is odd
245         prime->d[0] |= 1;
```

```
246 }
```

10.2.14.1.7 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.

```
247 TPM_RC
248 BnGeneratePrimeForRSA(
249     bigNum          prime,          // IN/OUT: points to the BN that will get the
250                                     //  random value
251     UINT32          bits,           // IN: number of bits to get
252     UINT32          exponent,       // IN: the exponent
253     RAND_STATE      *rand           // IN: the random state
254     )
255 {
256     BOOL            found = FALSE;
257 //
258     // Make sure that the prime is large enough
259     pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));
260     // Only try to handle specific sizes of keys in order to save overhead
261     pAssert((bits % 32) == 0);
262
263     prime->size = BITS_TO_CRYPT_WORDS(bits);
264
265     while(!found)
266     {
267 // The change below is to make sure that all keys that are generated from the same
268 // seed value will be the same regardless of the endianess or word size of the CPU.
269 //       DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits));// old
270 //       if(g_inFailureMode)                                                // old
271         if(!BnGetRandomBits(prime, bits, rand))                            // new
272             return TPM_RC_FAILURE;
273         RsaAdjustPrimeCandidate(prime);
274         found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;
275     }
276     return TPM_RC_SUCCESS;
277 }
278 #endif // ALG_RSA
```

10.2.15 CryptPrimeSieve.c

10.2.15.1 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;

//** Functions

//*** 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;

}
```

10.2.15.1.1 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
RsaNextPrime(
    uint32_t    lastPrime
```

```
    )
{
    if(lastPrime == 0)
        return 0;
    lastPrime >>= 1;
    for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
    {
        if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
            return ((lastPrime << 1) + 1);
    }
    return 0;
}
```

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.

```
const BYTE   seedValues[] = {
    0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
    0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
    0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
    0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
    0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
    0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
    0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
    0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
    0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
    0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
    0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
    0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
    0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
    0xd1};

#define USE_NIBBLE

#ifndef USE_NIBBLE
static const BYTE bitsInByte[256] = {
    0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
```

```
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
};
#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
```

10.2.15.1.2 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;
}
```

10.2.15.1.3 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;
```

```
153         sel = a[i];
154         // Subtract the bits in the last byte added.
155         sum -= BitsInByte(sel);
156         // Now process the byte, one bit at a time.
157         for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
158             sum += (sel & 1) != 0;
159         return (sum == n) ? retValue : -1;
160     }
161     typedef struct
162     {
163         UINT16      prime;
164         UINT16      count;
165     } SIEVE_MARKS;
166     const SIEVE_MARKS sieveMarks[5] = {
167         {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};
168
169     //*** PrimeSieve()
170     // This function does a prime sieve over the input 'field' which has as its
171     // starting address the value in bnN. Since this initializes the Sieve
172     // using a precomputed field with the bits associated with 3, 5 and 7 already
173     // turned off, the value of pnN may need to be adjusted by a few counts to allow
174     // the precomputed field to be used without modification.
175     //
176     // To get better performance, one could address the issue of developing the
177     // composite numbers. When the size of the prime gets large, the time for doing
178     // the divisions goes up, noticeably. It could be better to develop larger composite
179     // numbers even if they need to be bigNum's themselves. The object would be to
180     // reduce the number of times that the large prime is divided into a few large
181     // divides and then use smaller divides to get to the final 16 bit (or smaller)
182     // remainders.
183     LIB_EXPORT UINT32
184     PrimeSieve(
185         bigNum           bnN,       // IN/OUT: number to sieve
186         UINT32           fieldSize, // IN: size of the field area in bytes
187         BYTE            *field      // IN: field
188         )
189     {
190         UINT32           i;
191         UINT32           j;
192         UINT32           fieldBits = fieldSize * 8;
193         UINT32           r;
194         BYTE            *pField;
195         INT32            iter;
196         UINT32           adjust;
197         UINT32           mark = 0;
198         UINT32           count = sieveMarks[0].count;
199         UINT32           stop = sieveMarks[0].prime;
200         UINT32           composite;
201         UINT32           pList[8];
202         UINT32           next;
203
204         pAssert(field != NULL && bnN != NULL);
205
206         // If the remainder is odd, then subtracting the value will give an even number,
207         // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
208         // the even remainder.
209         adjust = (UINT32)BnModWord(bnN, 105);
210         if(adjust & 1)
211             adjust += 105;
212
213         // Adjust the input number so that it points to the first number in a
214         // aligned field.
215         BnSubWord(bnN, bnN, adjust);
216     //    pAssert(BnModWord(bnN, 105) == 0);
217         pField = field;
218         for(i = fieldSize; i >= sizeof(seedValues);
```

```
219             pField += sizeof(seedValues), i -= sizeof(seedValues))
220         {
221             memcpy(pField, seedValues, sizeof(seedValues));
222         }
223         if(i != 0)
224             memcpy(pField, seedValues, i);
225 
226         // Cycle through the primes, clearing bits
227         // Have already done 3, 5, and 7
228         iter = 7;
229 
230     #define NEXT_PRIME(iter)    (iter = RsaNextPrime(iter))
231         // Get the next N primes where N is determined by the mark in the sieveMarks
232         while((composite = NEXT_PRIME(iter)) != 0)
233         {
234             next = 0;
235             i = count;
236             pList[i--] = composite;
237             for(; i > 0; i--)
238             {
239                 next = NEXT_PRIME(iter);
240                 pList[i] = next;
241                 if(next != 0)
242                     composite *= next;
243             }
244             // Get the remainder when dividing the base field address
245             // by the composite
246             composite = (UINT32)BnModWord(bnN, composite);
247             // 'composite' is divisible by the composite components. for each of the
248             // composite components, divide 'composite'. That remainder (r) is used to
249             // pick a starting point for clearing the array. The stride is equal to the
250             // composite component. Note, the field only contains odd numbers. If the
251             // field were expanded to contain all numbers, then half of the bits would
252             // have already been cleared. We can save the trouble of clearing them a
253             // second time by having a stride of 2*next. Or we can take all of the even
254             // numbers out of the field and use a stride of 'next'
255             for(i = count; i > 0; i--)
256             {
257                 next = pList[i];
258                 if(next == 0)
259                     goto done;
260                 r = composite % next;
261             // these computations deal with the fact that we have picked a field-sized
262             // range that is aligned to a 105 count boundary. The problem is, this field
263             // only contains odd numbers. If we take our prime guess and walk through all
264             // the numbers using that prime as the 'stride', then every other 'stride' is
265             // going to be an even number. So, we are actually counting by 2 * the stride
266             // We want the count to start on an odd number at the start of our field. That
267             // is, we want to assume that we have counted up to the edge of the field by
268             // the 'stride' and now we are going to start flipping bits in the field as we
269             // continue to count up by 'stride'. If we take the base of our field and
270             // divide by the stride, we find out how much we find out how short the last
271             // count was from reaching the edge of the bit field. Say we get a quotient of
272             // 3 and remainder of 1. This means that after 3 strides, we are 1 short of
273             // the start of the field and the next stride will either land within the
274             // field or step completely over it. The confounding factor is that our field
275             // only contains odd numbers and our stride is actually 2 * stride. If the
276             // quoitent is even, then that means that when we add 2 * stride, we are going
277             // to hit another even number. So, we have to know if we need to back off
278             // by 1 stride before we start couting by 2 * stride.
279             // We can tell from the remainder whether we are on an even or odd
280             // stride when we hit the beginning of the table. If we are on an odd stride
281             // (r & 1), we would start half a stride in (next - r)/2. If we are on an
282             // even stride, we need 0.5 strides (next - r/2) because the table only has
283             // odd numbers. If the remainder happens to be zero, then the start of the
284             // table is on stride so no adjustment is necessary.
```

```
285             if(r & 1)           j = (next - r) / 2;
286             else if(r == 0)     j = 0;
287             else                 j = next - (r / 2);
288             for(; j < fieldBits; j += next)
289                 ClearBit(j, field, fieldSize);
290         }
291         if(next >= stop)
292         {
293             mark++;
294             count = sieveMarks[mark].count;
295             stop = sieveMarks[mark].prime;
296         }
297     }
298 done:
299     INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
300     i = BitsInArray(field, fieldSize);
301     INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
302     INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
303     return i;
304 }
305 #ifdef SIEVE_DEBUG
306 static uint32_t fieldSize = 210;
307
308 //***SetFieldSize()
309 // Function to set the field size used for prime generation. Used for tuning.
310 LIB_EXPORT uint32_t
311 SetFieldSize(
312     uint32_t         newFieldSize
313     )
314 {
315     if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
316         fieldSize = MAX_FIELD_SIZE;
317     else
318         fieldSize = newFieldSize;
319     return fieldSize;
320 }
321 #endif // SIEVE_DEBUG
```

10.2.15.1.4 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_FAILURE	TPM in failure mode, probably due to entropy source
TPM_RC_SUCCESS	candidate is probably prime
TPM_RC_NO_RESULT	candidate is not prime and couldn't find and alternative in the field

```
322 LIB_EXPORT TPM_RC
323 PrimeSelectWithSieve(
324     bigNum           candidate,         // IN/OUT: The candidate to filter
325     UINT32           e,                 // IN: the exponent
326     RAND_STATE      *rand               // IN: the random number generator state
327     )
328 {
329     BYTE             field[MAX_FIELD_SIZE];
330     UINT32           first;
```

```
331         UINT32          ones;
332         INT32           chosen;
333         BN_PRIME(test);
334         UINT32          modE;
335     #ifndef SIEVE_DEBUG
336         UINT32          fieldSize = MAX_FIELD_SIZE;
337     #endif
338         UINT32          primeSize;
339     //
340         // Adjust the field size and prime table list to fit the size of the prime
341         // being tested. This is done to try to optimize the trade-off between the
342         // dividing done for sieving and the time for Miller-Rabin. When the size
343         // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
344         // cost of the sieving. However, the time for Miller-Rabin goes up considerably
345         // faster than the cost of dividing by a number of primes.
346         primeSize = BnSizeInBits(candidate);
347
348         if(primeSize <= 512)
349         {
350             RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
351         }
352         else if(primeSize <= 1024)
353         {
354             RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
355         }
356         else
357         {
358             RsaAdjustPrimeLimit(0);     // Use all available
359         }
360
361         // Save the low-order word to use as a search generator and make sure that
362         // it has some interesting range to it
363         first = (UINT32)(candidate->d[0] | 0x80000000);
364
365         // Sieve the field
366         ones = PrimeSieve(candidate, fieldSize, field);
367         pAssert(ones > 0 && ones < (fieldSize * 8));
368         for(; ones > 0; ones--)
369         {
370             // Decide which bit to look at and find its offset
371             chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
372
373             if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
374                 FAIL(FATAL_ERROR_INTERNAL);
375
376             // Set this as the trial prime
377             BnAddWord(test, candidate, (crypt_uword_t)(chosen * 2));
378
379             // The exponent might not have been one of the tested primes so
380             // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
381             // Note: This is the same as 1 != p mod e
382             modE = (UINT32)BnModWord(test, e);
383             if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
384             {
385                 BnCopy(candidate, test);
386                 return TPM_RC_SUCCESS;
387             }
388             // Clear the bit just tested
389             ClearBit(chosen, field, fieldSize);
390         }
391         // Ran out of bits and couldn't find a prime in this field
392         INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
393         return (g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_NO_RESULT);
394     }
395     #if RSA_INSTRUMENT
396     static char             a[256];
```

```
397
398 //*** PrintTuple()
399 char *
400 PrintTuple(
401     UINT32      *i
402     )
403 {
404     sprintf(a, "{%d, %d, %d}", i[0], i[1], i[2]);
405     return a;
406 }
407
408 #define CLEAR_VALUE(x)    memset(x, 0, sizeof(x))
409
410 //*** RsaSimulationEnd()
411 void
412 RsaSimulationEnd(
413     void
414     )
415 {
416     int         i;
417     UINT32      averages[3];
418     UINT32      nonFirst = 0;
419     if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
420     {
421         printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
422         printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
423         printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
424         printf("Primes checked with Miller-Rabin = %s\n",
425                PrintTuple(MillerRabinTrials));
426         for(i = 0; i < 3; i++)
427             averages[i] = (totalFieldsSieved[i]
428                            != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
429                            : 0);
430         printf("Average candidates in field %s\n", PrintTuple(averages));
431         for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
432         i++)
433             nonFirst += failedAtIteration[i];
434         printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
435
436     }
437     CLEAR_VALUE(PrimeCounts);
438     CLEAR_VALUE(totalFieldsSieved);
439     CLEAR_VALUE(noPrimeFields);
440     CLEAR_VALUE(MillerRabinTrials);
441     CLEAR_VALUE(bitsInFieldAfterSieve);
442 }
443
444 //*** GetSieveStats()
445 LIB_EXPORT void
446 GetSieveStats(
447     uint32_t        *trials,
448     uint32_t        *emptyFields,
449     uint32_t        *averageBits
450     )
451 {
452     uint32_t        totalBits;
453     uint32_t        fields;
454     *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
455     *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
456     fields = totalFieldsSieved[0] + totalFieldsSieved[1]
457         + totalFieldsSieved[2];
458     totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
459         + bitsInFieldAfterSieve[2];
460     if(fields != 0)
461         *averageBits = totalBits / fields;
462     else
```

```
463             *averageBits = 0;
464         CLEAR_VALUE(PrimeCounts);
465         CLEAR_VALUE(totalFieldsSieved);
466         CLEAR_VALUE(noPrimeFields);
467         CLEAR_VALUE(MillerRabinTrials);
468         CLEAR_VALUE(bitsInFieldAfterSieve);
469
470     }
471     #endif
472
473     #endif // RSA_KEY_SIEVE
474
475     #if !RSA_INSTRUMENT
476
477     //*** RsaSimulationEnd()
478     // Stub for call when not doing instrumentation.
479     void
480     RsaSimulationEnd(
481         void
482         )
483     {
484         return;
485     }
486     #endif
```

10.2.16 CryptRand.c

10.2.16.1 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.

```
1   #include "Tpm.h"
```

Pull in the test vector definitions and define the space

```
2   #include    "PRNG_TestVectors.h"
3   const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
4   const BYTE DRBG_NistTestVector_GeneratedInterm[] =
5                                   {DRBG_TEST_GENERATED_INTERM};
6
7   const BYTE DRBG_NistTestVector_EntropyReseed[] =
8                                   {DRBG_TEST_RESEED_ENTROPY};
9   const BYTE DRBG_NistTestVector_Generated[] = {DRBG_TEST_GENERATED};
10
11  //** Derivation Functions
12  //*** Description
13  // The functions in this section are used to reduce the personalization input values
14  // to make them usable as input for reseeding and instantiation. The overall
15  // behavior is intended to produce the same results as described in SP800-90A,
16  // section 10.4.2 "Derivation Function Using a Block Cipher Algorithm
17  // (Block_Cipher_df)." The code is broken into several subroutines to deal with the
18  // fact that the data used for personalization may come in several separate blocks
19  // such as a Template hash and a proof value and a primary seed.
20
21  //*** Derivation Function Defines and Structures
22
23  #define     DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
24  #if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
25  #   error "CryptRand.c only written for AES with 128- or 256-bit keys."
26  #endif
```

```
27  typedef struct
28  {
29      DRBG_KEY_SCHEDULE   keySchedule;
30      DRBG_IV             iv[DF_COUNT];
31      DRBG_IV             out1;
32      DRBG_IV             buf;
33      int                 contents;
34  } DF_STATE, *PDF_STATE;
```

10.2.16.1.1 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.

```
35  static void
36  DfCompute(
37      PDF_STATE        dfState
38      )
39  {
40      int              i;
41      int              iv;
42      crypt_uword_t   *pIv;
43      crypt_uword_t    temp[DRBG_IV_SIZE_WORDS] = {0};
44  //
45      for(iv = 0; iv < DF_COUNT; iv++)
46      {
47          pIv = (crypt_uword_t *)&dfState->iv[iv].words[0];
48          for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
49          {
50              temp[i] ^= pIv[i] ^ dfState->buf.words[i];
51          }
52          DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
53      }
54      for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
55          dfState->buf.words[i] = 0;
56      dfState->contents = 0;
57  }
```

10.2.16.1.2 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```
58  static void
59  DfStart(
60      PDF_STATE        dfState,
61      uint32_t         inputLength
62      )
63  {
64      BYTE            init[8];
65      int             i;
66      UINT32          drbgSeedSize = sizeof(DRBG_SEED);
67
68      const BYTE dfKey[DRBG_KEY_SIZE_BYTES] = {
69          0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
70          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
71      #if DRBG_KEY_SIZE_BYTES > 16
72          ,0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
73          0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
74      #endif
75      };
76      memset(dfState, 0, sizeof(DF_STATE));
```

```
77      DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
78      // Create the first chaining values
79      for(i = 0; i < DF_COUNT; i++)
80          ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
81      DfCompute(dfState);
82      // initialize the first 64 bits of the IV in a way that doesn't depend
83      // on the size of the words used.
84      UINT32_TO_BYTE_ARRAY(inputLength, init);
85      UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
86      memcpy(&dfState->iv[0], init, 8);
87      dfState->contents = 4;
88  }
```

10.2.16.1.3 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().

```
89  static void
90  DfUpdate(
91      PDF_STATE        dfState,
92      int              size,
93      const BYTE      *data
94      )
95  {
96      while(size > 0)
97      {
98          int         toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
99          if(size < toFill)
100             toFill = size;
101         // Copy as many bytes as there are or until the state buffer is full
102         memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
103         // Reduce the size left by the amount copied
104         size -= toFill;
105         // Advance the data pointer by the amount copied
106         data += toFill;
107         // increase the buffer contents count by the amount copied
108         dfState->contents += toFill;
109         pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
110         // If we have a full buffer, do a computation pass.
111         if(dfState->contents == DRBG_IV_SIZE_BYTES)
112             DfCompute(dfState);
113     }
114 }
```

10.2.16.1.4 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.

```
115 static DRBG_SEED *
116 DfEnd(
117     PDF_STATE        dfState
118     )
119 {
120     // Since DfCompute is always called when a buffer is full, there is always
121     // space in the buffer for the terminator
122     dfState->buf.bytes[dfState->contents++] = 0x80;
123     // If the buffer is not full, pad with zeros
124     while(dfState->contents < DRBG_IV_SIZE_BYTES)
125         dfState->buf.bytes[dfState->contents++] = 0;
```

```
126         // Do a final state update
127         DfCompute(dfState);
128         return (DRBG_SEED *)&dfState->iv;
129     }
```

10.2.16.1.5 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();

```
130     static DRBG_SEED *
131     DfBuffer(
132         DRBG_SEED       *output,        // OUT: receives the result
133         int              size,          // IN: size of the buffer to add
134         BYTE            *buf            // IN: address of the buffer
135         )
136     {
137         DF_STATE        dfState;
138         if(size == 0 || buf == NULL)
139             return NULL;
140         // Initialize the derivation function
141         DfStart(&dfState, size);
142         DfUpdate(&dfState, size, buf);
143         DfEnd(&dfState);
144         memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
145         return output;
146     }
```

10.2.16.1.6 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

```
147     BOOL
148     DRBG_GetEntropy(
149         UINT32           requiredEntropy,   // IN: requested number of bytes of full
150                                             //     entropy
151         BYTE            *entropy            // OUT: buffer to return collected entropy
152         )
153     {
154     #if !USE_DEBUG_RNG
155
156         UINT32       obtainedEntropy;
157         INT32        returnedEntropy;
158
159     // If in debug mode, always use the self-test values for initialization
160         if(IsSelfTest())
161         {
162     #endif
163             // If doing simulated DRBG, then check to see if the
164             // entropyFailure condition is being tested
165             if(!IsEntropyBad())
166             {
167                 // In self-test, the caller should be asking for exactly the seed
168                 // size of entropy.
```

```
169             pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
170             memcpy(entropy, DRBG_NistTestVector_Entropy,
171                    sizeof(DRBG_NistTestVector_Entropy));
172         }
173 #if !USE_DEBUG_RNG
174     }
175     else if(!IsEntropyBad())
176     {
177         // Collect entropy
178         // Note: In debug mode, the only "entropy" value ever returned
179         // is the value of the self-test vector.
180         for(returnedEntropy = 1, obtainedEntropy = 0;
181             obtainedEntropy < requiredEntropy && !IsEntropyBad();
182             obtainedEntropy += returnedEntropy)
183         {
184             returnedEntropy = _plat__GetEntropy(&entropy[obtainedEntropy],
185                                                 requiredEntropy - obtainedEntropy);
186             if(returnedEntropy <= 0)
187                 SetEntropyBad();
188         }
189     }
190 #endif
191     return !IsEntropyBad();
192 }
```

10.2.16.1.7 IncrementIv()

This function increments the IV value by 1. It is used by EncryptDRBG().

```
193 void
194 IncrementIv(
195     DRBG_IV         *iv
196     )
197 {
198     BYTE      *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
199     while((--ivP >= (BYTE *)iv) && ((*ivP = ((*ivP + 1) & 0xFF)) == 0));
200 }
```

10.2.16.1.8 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.

```
201 static BOOL
202 EncryptDRBG(
203     BYTE                *dOut,
204     UINT32               dOutBytes,
205     DRBG_KEY_SCHEDULE   *keySchedule,
206     DRBG_IV             *iv,
207     UINT32              *lastValue      // Points to the last output value
208     )
209 {
210 #if FIPS_COMPLIANT
211 // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
212 // no two consecutive values are the same. This overhead is not incurred if the TPM
213 // is not required to be FIPS compliant
214 //
215     UINT32           temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
216     int              i;
217     BYTE            *p;
218
219     for(; dOutBytes > 0;)
220     {
```

```
221             // Increment the IV before each encryption (this is what makes this
222             // different from normal counter-mode encryption
223             IncrementIv(iv);
224             DRBG_ENCRYPT(keySchedule, iv, temp);
225     // Expect a 16 byte block
226     #if DRBG_IV_SIZE_BITS != 128
227     #error  "Unsuppored IV size in DRBG"
228     #endif
229             if((lastValue[0] == temp[0])
230                 && (lastValue[1] == temp[1])
231                 && (lastValue[2] == temp[2])
232                 && (lastValue[3] == temp[3])
233                 )
234             {
235                 LOG_FAILURE(FATAL_ERROR_ENTROPY);
236                 return FALSE;
237             }
238             lastValue[0] = temp[0];
239             lastValue[1] = temp[1];
240             lastValue[2] = temp[2];
241             lastValue[3] = temp[3];
242             i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
243             dOutBytes -= i;
244             for(p = (BYTE *)temp; i > 0; i--)
245                 *dOut++ = *p++;
246         }
247     #else // version without continuous self-test
248         NOT_REFERENCED(lastValue);
249         for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
250         dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
251         {
252             // Increment the IV
253             IncrementIv(iv);
254             DRBG_ENCRYPT(keySchedule, iv, dOut);
255         }
256         // If there is a partial, generate into a block-sized
257         // temp buffer and copy to the output.
258         if(dOutBytes != 0)
259         {
260             BYTE        temp[DRBG_IV_SIZE_BYTES];
261             // Increment the IV
262             IncrementIv(iv);
263             DRBG_ENCRYPT(keySchedule, iv, temp);
264             memcpy(dOut, temp, dOutBytes);
265         }
266     #endif
267         return TRUE;
268     }
```

10.2.16.1.9 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.

```
269     static BOOL
270     DRBG_Update(
271         DRBG_STATE          *drbgState,     // IN:OUT state to update
272         DRBG_KEY_SCHEDULE   *keySchedule,   // IN: the key schedule (optional)
```

```
273         DRBG_SEED           *providedData   // IN: additional data
274         )
275     {
276         UINT32               i;
277         BYTE                *temp = (BYTE *)&drbgState->seed;
278         DRBG_KEY            *key = pDRBG_KEY(&drbgState->seed);
279         DRBG_IV             *iv = pDRBG_IV(&drbgState->seed);
280         DRBG_KEY_SCHEDULE    localKeySchedule;
281     //
282         pAssert(drbgState->magic == DRBG_MAGIC);
283
284         // If an key schedule was not provided, make one
285         if(keySchedule == NULL)
286         {
287             if(DRBG_ENCRYPT_SETUP((BYTE *)key,
288                 DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
289             {
290                 LOG_FAILURE(FATAL_ERROR_INTERNAL);
291                 return FALSE;
292             }
293             keySchedule = &localKeySchedule;
294         }
295         // Encrypt the temp value
296
297         EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
298                     drbgState->lastValue);
299         if(providedData != NULL)
300         {
301             BYTE        *pP = (BYTE *)providedData;
302             for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
303                 *temp++ ^= *pP++;
304         }
305         // Since temp points to the input key and IV, we are done and
306         // don't need to copy the resulting 'temp' to drbgState->seed
307         return TRUE;
308     }
```

10.2.16.1.10 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

```
309     BOOL
310     DRBG_Reseed(
311         DRBG_STATE          *drbgState,         // IN: the state to update
312         DRBG_SEED           *providedEntropy,   // IN: entropy
313         DRBG_SEED           *additionalData     // IN:
314         )
315     {
316         DRBG_SEED            seed;
317
318         pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
319
320         if(providedEntropy == NULL)
321         {
322             providedEntropy = &seed;
```

```
323             if(!DRBG_GetEntropy(sizeof(DRBG_SEED), (BYTE *)providedEntropy))
324                 return FALSE;
325         }
326         if(additionalData != NULL)
327         {
328             unsigned int          i;
329
330             // XOR the provided data into the provided entropy
331             for(i = 0; i < sizeof(DRBG_SEED); i++)
332                 ((BYTE *)providedEntropy)[i] ^= ((BYTE *)additionalData)[i];
333         }
334         DRBG_Update(drbgState, NULL, providedEntropy);
335
336         drbgState->reseedCounter = 1;
337
338         return TRUE;
339     }
```

10.2.16.1.11 DRBG_SelfTest()

This is run when the DRBG is instantiated and at startup

Return Value	Meaning
TRUE(1)	test OK
FALSE(0)	test failed

```
340     BOOL
341     DRBG_SelfTest(
342         void
343         )
344     {
345         BYTE             buf[sizeof(DRBG_NistTestVector_Generated)];
346         DRBG_SEED        seed;
347         UINT32           i;
348         BYTE            *p;
349         DRBG_STATE       testState;
350     //
351         pAssert(!IsSelfTest());
352
353         SetSelfTest();
354         SetDrbgTested();
355         // Do an instantiate
356         if(!DRBG_Instantiate(&testState, 0, NULL))
357             return FALSE;
358     #if DRBG_DEBUG_PRINT
359         dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
360                         "Key after Instantiate");
361         dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
362                         "Value after Instantiate");
363     #endif
364         if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
365             return FALSE;
366     #if DRBG_DEBUG_PRINT
367         dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
368                         "Key after 1st Generate");
369         dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
370                         "Value after 1st Generate");
371     #endif
372         if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
373             return FALSE;
374         memcpy(seed.bytes, DRBG_NistTestVector_EntropyReseed, sizeof(seed));
375         DRBG_Reseed(&testState, &seed, NULL);
```

```
376 #if DRBG_DEBUG_PRINT
377     dbgDumpMemBlock((BYTE *)pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
378                     "Key after 2nd Generate");
379     dbgDumpMemBlock((BYTE *)pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
380                     "Value after 2nd Generate");
381     dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
382 #endif
383     if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
384         return FALSE;
385     if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
386         return FALSE;
387     ClearSelfTest();
388 
389     DRBG_Uninstantiate(&testState);
390     for(p = (BYTE *)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
391     {
392         if(*p++)
393             return FALSE;
394     }
395     // Simulate hardware failure to make sure that we get an error when
396     // trying to instantiate
397     SetEntropyBad();
398     if(DRBG_Instantiate(&testState, 0, NULL))
399        return FALSE;
400     ClearEntropyBad();
401 
402     return TRUE;
403 }
```

10.2.16.2 Public Interface

10.2.16.2.1 Description

The functions in this section are the interface to the RNG. These are the functions that are used by TPM.lib.

10.2.16.2.2 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

```
404 LIB_EXPORT TPM_RC
405 CryptRandomStir(
406     UINT16           additionalDataSize,
407     BYTE            *additionalData
408     )
409 {
410 #if !USE_DEBUG_RNG
411     DRBG_SEED        tmpBuf;
412     DRBG_SEED        dfResult;
413 //
414     // All reseed with outside data starts with a buffer full of entropy
415     if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))
416         return TPM_RC_NO_RESULT;
417 
418     DRBG_Reseed(&drbgDefault, &tmpBuf,
419                 DfBuffer(&dfResult, additionalDataSize, additionalData));
420     drbgDefault.reseedCounter = 1;
```

```
421
422         return TPM_RC_SUCCESS;
423
424     #else
425         // If doing debug, use the input data as the initial setting for the RNG state
426         // so that the test can be reset at any time.
427         // Note: If this is called with a data size of 0 or less, nothing happens. The
428         // presumption is that, in a debug environment, the caller will have specific
429         // values for initialization, so this check is just a simple way to prevent
430         // inadvertent programming errors from screwing things up. This doesn't use an
431         // pAssert() because the non-debug version of this function will accept these
432         // parameters as meaning that there is no additionalData and only hardware
433         // entropy is used.
434         if((additionalDataSize > 0) && (additionalData != NULL))
435         {
436             memset(drbgDefault.seed.bytes, 0, sizeof(drbgDefault.seed.bytes));
437             memcpy(drbgDefault.seed.bytes, additionalData,
438                    MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
439         }
440         drbgDefault.reseedCounter = 1;
441
442         return TPM_RC_SUCCESS;
443     #endif
444     }
```

10.2.16.2.3 CryptRandomGenerate()

Generate a *randomSize* number or random bytes.

```
445     LIB_EXPORT UINT16
446     CryptRandomGenerate(
447         UINT16           randomSize,
448         BYTE            *buffer
449         )
450     {
451         return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, randomSize);
452     }
```

10.2.16.2.4 DRBG_InstantiateSeededKdf()

This function is used to instantiate a KDF-based RNG. This is used for derivations. This function always returns TRUE.

```
453     LIB_EXPORT BOOL
454     DRBG_InstantiateSeededKdf(
455         KDF_STATE       *state,         // OUT: buffer to hold the state
456         TPM_ALG_ID       hashAlg,       // IN: hash algorithm
457         TPM_ALG_ID       kdf,           // IN: the KDF to use
458         TPM2B           *seed,          // IN: the seed to use
459         const TPM2B     *label,         // IN: a label for the generation process.
460         TPM2B           *context,       // IN: the context value
461         UINT32           limit          // IN: Maximum number of bits from the KDF
462         )
463     {
464         state->magic = KDF_MAGIC;
465         state->limit = limit;
466         state->seed = seed;
467         state->hash = hashAlg;
468         state->kdf = kdf;
469         state->label = label;
470         state->context = context;
471         state->digestSize = CryptHashGetDigestSize(hashAlg);
472         state->counter = 0;
```

```
473         state->residual.t.size = 0;
474         return TRUE;
475     }
```

10.2.16.2.5 DRBG_AdditionalData()

Function to reseed the DRBG with additional entropy. This is normally called before computing the protection value of a primary key in the Endorsement hierarchy.

```
476     LIB_EXPORT void
477     DRBG_AdditionalData(
478         DRBG_STATE      *drbgState,     // IN:OUT state to update
479         TPM2B           *additionalData // IN: value to incorporate
480         )
481     {
482         DRBG_SEED        dfResult;
483         if(drbgState->magic == DRBG_MAGIC)
484         {
485             DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
486             DRBG_Reseed(drbgState, &dfResult, NULL);
487         }
488     }
```

10.2.16.2.6 DRBG_InstantiateSeeded()

This function is used to instantiate a random number generator from seed values. The nominal use of this generator is to create sequences of pseudo-random numbers from a seed value.

Error Returns	Meaning
TPM_RC_FAILURE	DRBG self-test failure

```
489     LIB_EXPORT TPM_RC
490     DRBG_InstantiateSeeded(
491         DRBG_STATE      *drbgState,     // IN/OUT: buffer to hold the state
492         const TPM2B     *seed,          // IN: the seed to use
493         const TPM2B     *purpose,       // IN: a label for the generation process.
494         const TPM2B     *name,          // IN: name of the object
495         const TPM2B     *additional     // IN: additional data
496         )
497     {
498         DF_STATE         dfState;
499         int              totalInputSize;
500         // DRBG should have been tested, but...
501         if(!IsDrbgTested() && !DRBG_SelfTest())
502         {
503             LOG_FAILURE(FATAL_ERROR_SELF_TEST);
504             return TPM_RC_FAILURE;
505         }
506         // Initialize the DRBG state
507         memset(drbgState, 0, sizeof(DRBG_STATE));
508         drbgState->magic = DRBG_MAGIC;
509
510         // Size all of the values
511         totalInputSize = (seed != NULL) ? seed->size : 0;
512         totalInputSize += (purpose != NULL) ? purpose->size : 0;
513         totalInputSize += (name != NULL) ? name->size : 0;
514         totalInputSize += (additional != NULL) ? additional->size : 0;
515
516         // Initialize the derivation
517         DfStart(&dfState, totalInputSize);
518
```

```
519         // Run all the input strings through the derivation function
520         if(seed != NULL)
521             DfUpdate(&dfState, seed->size, seed->buffer);
522         if(purpose != NULL)
523             DfUpdate(&dfState, purpose->size, purpose->buffer);
524         if(name != NULL)
525             DfUpdate(&dfState, name->size, name->buffer);
526         if(additional != NULL)
527             DfUpdate(&dfState, additional->size, additional->buffer);
528 
529         // Used the derivation function output as the "entropy" input. This is not
530         // how it is described in SP800-90A but this is the equivalent function
531         DRBG_Reseed(((DRBG_STATE *)drbgState), DfEnd(&dfState), NULL);
532 
533         return TPM_RC_SUCCESS;
534     }
```

10.2.16.2.7 CryptRandStartup()

This function is called when TPM_Startup is executed. This function always returns TRUE.

```
535     LIB_EXPORT BOOL
536     CryptRandStartup(
537         void
538         )
539     {
540     #if ! _DRBG_STATE_SAVE
541         // If not saved in NV, re-instantiate on each startup
542         DRBG_Instantiate(&drbgDefault, 0, NULL);
543     #else
544         // If the running state is saved in NV, NV has to be loaded before it can
545         // be updated
546         if(go.drbgState.magic == DRBG_MAGIC)
547             DRBG_Reseed(&go.drbgState, NULL, NULL);
548         else
549             DRBG_Instantiate(&go.drbgState, 0, NULL);
550     #endif
551         return TRUE;
552     }
```

10.2.16.2.7.1 CryptRandInit()

This function is called when _TPM_Init() is being processed.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
LIB_EXPORT BOOL
CryptRandInit(
    void
    )
{
#if !USE_DEBUG_RNG
    _plat__GetEntropy(NULL, 0);
#endif
    return DRBG_SelfTest();
}
```

10.2.16.2.8 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 zero without generating any bits or updating the reseed counter. This function returns the number of bytes produced which could be less than the number requested if the request is too large ("too large" is implementation dependent.)

```
LIB_EXPORT UINT16
DRBG_Generate(
    RAND_STATE      *state,
    BYTE            *random,        // OUT: buffer to receive the random values
    UINT16           randomSize     // IN: the number of bytes to generate
    )
{
    if(state == NULL)
        state = (RAND_STATE *)&drbgDefault;
    if(random == NULL)
        return 0;

    // If the caller used a KDF state, generate a sequence from the KDF not to
    // exceed the limit.
    if(state->kdf.magic == KDF_MAGIC)
    {
        KDF_STATE       *kdf = (KDF_STATE *)state;
        UINT32           counter = (UINT32)kdf->counter;
        INT32            bytesLeft = randomSize;
//
        // If the number of bytes to be returned would put the generator
        // over the limit, then return 0
        if((((kdf->counter * kdf->digestSize) + randomSize) * 8) > kdf->limit)
            return 0;
        // Process partial and full blocks until all requested bytes provided
        while(bytesLeft > 0)
        {
            // If there is any residual data in the buffer, copy it to the output
            // buffer
            if(kdf->residual.t.size > 0)
            {
                INT32      size;
//
```

```
596                 // Don't use more of the residual than will fit or more than are
597                 // available
598                 size = MIN(kdf->residual.t.size, bytesLeft);
599 
600                 // Copy some or all of the residual to the output. The residual is
601                 // at the end of the buffer. The residual might be a full buffer.
602                 MemoryCopy(random,
603                            &kdf->residual.t.buffer
604                            [kdf->digestSize - kdf->residual.t.size], size);
605 
606                 // Advance the buffer pointer
607                 random += size;
608 
609                 // Reduce the number of bytes left to get
610                 bytesLeft -= size;
611 
612                 // And reduce the residual size appropriately
613                 kdf->residual.t.size -= (UINT16)size;
614             }
615             else
616             {
617                 UINT16           blocks = (UINT16)(bytesLeft / kdf->digestSize);
618 //
619                 // Get the number of required full blocks
620                 if(blocks > 0)
621                 {
622                     UINT16      size = blocks * kdf->digestSize;
623 // Get some number of full blocks and put them in the return buffer
624                     CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
625                               kdf->limit, random, &counter, blocks);
626 
627                     // reduce the size remaining to be moved and advance the pointer
628                     bytesLeft -= size;
629                     random += size;
630                 }
631                 else
632                 {
633                     // Fill the residual buffer with a full block and then loop to
634                     // top to get part of it copied to the output.
635                     kdf->residual.t.size = CryptKDFa(kdf->hash, kdf->seed,
636                                                      kdf->label, kdf->context, NULL,
637                                                      kdf->limit,
638                                                      kdf->residual.t.buffer,
639                                                      &counter, 1);
640                 }
641             }
642         }
643         kdf->counter = counter;
644         return randomSize;
645     }
646     else if(state->drbg.magic == DRBG_MAGIC)
647     {
648         DRBG_STATE          *drbgState = (DRBG_STATE *)state;
649         DRBG_KEY_SCHEDULE    keySchedule;
650         DRBG_SEED           *seed = &drbgState->seed;
651 
652         if(drbgState->reseedCounter >= CTR_DRBG_MAX_REQUESTS_PER_RESEED)
653         {
654             if(drbgState == &drbgDefault)
655             {
656                 DRBG_Reseed(drbgState, NULL, NULL);
657                 if(IsEntropyBad() && !IsSelfTest())
658                     return 0;
659             }
660             else
661             {
```

```
662                 // If this is a PRNG then the only way to get
663                 // here is if the SW has run away.
664                 LOG_FAILURE(FATAL_ERROR_INTERNAL);
665                 return 0;
666             }
667         }
668         // if the allowed number of bytes in a request is larger than the
669         // less than the number of bytes that can be requested, then check
670 #if UINT16_MAX >=  CTR_DRBG_MAX_BYTES_PER_REQUEST
671         if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
672             randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
673 #endif
674         // Create  encryption schedule
675         if(DRBG_ENCRYPT_SETUP((BYTE *)pDRBG_KEY(seed),
676                               DRBG_KEY_SIZE_BITS, &keySchedule) != 0)
677         {
678             LOG_FAILURE(FATAL_ERROR_INTERNAL);
679             return 0;
680         }
681         // Generate the random data
682         EncryptDRBG(random, randomSize, &keySchedule, pDRBG_IV(seed),
683                     drbgState->lastValue);
684         // Do a key update
685         DRBG_Update(drbgState, &keySchedule, NULL);
686
687         // Increment the reseed counter
688         drbgState->reseedCounter += 1;
689     }
690     else
691     {
692         LOG_FAILURE(FATAL_ERROR_INTERNAL);
693         return FALSE;
694     }
695     return randomSize;
696 }
```

10.2.16.2.9 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

```
697 LIB_EXPORT BOOL
698 DRBG_Instantiate(
699     DRBG_STATE      *drbgState,         // OUT: the instantiated value
700     UINT16           pSize,             // IN: Size of personalization string
701     BYTE            *personalization    // IN: The personalization string
702     )
703 {
704     DRBG_SEED        seed;
705     DRBG_SEED        dfResult;
706 //
707     pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
708     // If the DRBG has not been tested, test when doing an instantiation. Since
709     // Instantiation is called during self test, make sure we don't get stuck in a
710     // loop.
711     if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
712         return FALSE;
713     // If doing a self test, DRBG_GetEntropy will return the NIST
```

```
714         // test vector value.
715         if(!DRBG_GetEntropy(sizeof(seed), (BYTE *)&seed))
716             return FALSE;
717         // set everything to zero
718         memset(drbgState, 0, sizeof(DRBG_STATE));
719         drbgState->magic = DRBG_MAGIC;
720
721         // Steps 1, 2, 3, 6, 7 of SP 800-90A 10.2.1.3.1 are exactly what
722         // reseeding does. So, do a reduction on the personalization value (if any)
723         // and do a reseed.
724         DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
725
726         return TRUE;
727     }
```

10.2.16.2.10 DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

Error Returns	Meaning
TPM_RC_VALUE	not a valid state

```
728     LIB_EXPORT TPM_RC
729     DRBG_Uninstantiate(
730         DRBG_STATE      *drbgState      // IN/OUT: working state to erase
731         )
732     {
733         if((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
734             return TPM_RC_VALUE;
735         memset(drbgState, 0, sizeof(DRBG_STATE));
736         return TPM_RC_SUCCESS;
737     }
```

10.2.17 CryptRsa.c

10.2.17.1 Introduction

This file contains implementation of cryptographic primitives for RSA. Vendors may replace the implementation in this file with their own library functions.

10.2.17.2 Includes

Need this define to get the *private* defines for this function

```
#define CRYPT_RSA_C
#include "Tpm.h"
#if ALG_RSA
```

10.2.17.3 Obligatory Initialization Functions

10.2.17.3.1 CryptRsaInit()

Function called at _TPM_Init().

```
BOOL
CryptRsaInit(
    void
    )
{
    return TRUE;
}
```

10.2.17.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```
BOOL
CryptRsaStartup(
    void
    )
{
    return TRUE;
}
```

10.2.17.4 Internal Functions

10.2.17.4.1 RsaInitializeExponent()

This function initializes the bignum data structure that holds the private exponent. This function returns the pointer to the private exponent value so that it can be used in an initializer for a data declaration.

```
static privateExponent *
RsaInitializeExponent(
    privateExponent       *Z
    )
{
    bigNum               *bn = (bigNum *)&Z->P;
    int                   i;
//
```

```
26      for(i = 0; i < 5; i++)
27      {
28          bn[i] = (bigNum)&Z->entries[i];
29          BnInit(bn[i], BYTES_TO_CRYPT_WORDS(sizeof(Z->entries[0].d)));
30      }
31      return Z;
32  }
```

10.2.17.4.2 MakePgreaterThanQ()

This function swaps the pointers for P and Q if Q happens to be larger than Q.

```
33  static void
34  MakePgreaterThanQ(
35      privateExponent         *Z
36  )
37  {
38      if(BnUnsignedCmp(Z->P, Z->Q) < 0)
39      {
40          bigNum          bnT = Z->P;
41          Z->P = Z->Q;
42          Z->Q = bnT;
43      }
44  }
```

10.2.17.4.3 PackExponent()

This function takes the bignum private exponent and converts it into TPM2B form. In this form, the size field contains the overall size of the packed data. The buffer contains 5, equal sized values in P, Q, *dP*, *dQ*, *qInv* order. For example, if a key has a 2Kb public key, then the packed private key will contain 5, 1Kb values. This form makes it relatively easy to load and save the values without changing the normal unmarshaling to do anything more than allow a larger TPM2B for the private key. Also, when exporting the value, all that is needed is to change the size field of the private key in order to save just the P value.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure // The data is too big to fit

```
45  static BOOL
46  PackExponent(
47      TPM2B_PRIVATE_KEY_RSA       *packed,
48      privateExponent             *Z
49  )
50  {
51      int                         i;
52      UINT16                      primeSize = (UINT16)BITS_TO_BYTES(BnMsb(Z->P));
53      UINT16                      pS = primeSize;
54  //
55      pAssert((primeSize * 5) <= sizeof(packed->t.buffer));
56      packed->t.size = (primeSize * 5) + RSA_prime_flag;
57      for(i = 0; i < 5; i++)
58          if(!BnToBytes((bigNum)&Z->entries[i], &packed->t.buffer[primeSize * i], &pS))
59              return FALSE;
60      if(pS != primeSize)
61          return FALSE;
62      return TRUE;
63  }
```

10.2.17.4.4 UnpackExponent()

This function unpacks the private exponent from its TPM2B form into its bignum form.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	TPM2B is not the correct size

```
64  static BOOL
65  UnpackExponent(
66      TPM2B_PRIVATE_KEY_RSA       *b,
67      privateExponent             *Z
68  )
69  {
70          UINT16                  primeSize = b->t.size & ~RSA_prime_flag;
71          int                     i;
72          bigNum                  *bn = &Z->P;
73  //
74      VERIFY(b->t.size & RSA_prime_flag);
75      RsaInitializeExponent(Z);
76      VERIFY((primeSize % 5) == 0);
77      primeSize /= 5;
78      for(i = 0; i < 5; i++)
79          VERIFY(BnFromBytes(bn[i], &b->t.buffer[primeSize * i], primeSize)
80                  != NULL);
81      MakePgreaterThanQ(Z);
82      return TRUE;
83  Error:
84      return FALSE;
85   }
```

10.2.17.4.5 ComputePrivateExponent()

This function computes the private exponent from the primes.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
86  static BOOL
87  ComputePrivateExponent(
88      bigNum                  pubExp,         // IN: the public exponent
89      privateExponent     *Z                  // IN/OUT: on input, has primes P and Q. On
90                                              //         output, has P, Q, dP, dQ, and pInv
91      )
92  {
93      BOOL                pOK;
94      BOOL                qOK;
95      BN_PRIME(pT);
96  //
97      // make p the larger value so that m2 is always less than p
98      MakePgreaterThanQ(Z);
99
100     //dP = (1/e) mod (p-1)
101     pOK = BnSubWord(pT, Z->P, 1);
102     pOK = pOK && BnModInverse(Z->dP, pubExp, pT);
103     //dQ = (1/e) mod (q-1)
104     qOK = BnSubWord(pT, Z->Q, 1);
105     qOK = qOK && BnModInverse(Z->dQ, pubExp, pT);
106     // qInv = (1/q) mod p
```

```
107         if(pOK && qOK)
108             pOK = qOK = BnModInverse(Z->qInv, Z->Q, Z->P);
109         if(!pOK)
110             BnSetWord(Z->P, 0);
111         if(!qOK)
112             BnSetWord(Z->Q, 0);
113         return pOK && qOK;
114     }
```

10.2.17.4.6 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

```
115     static BOOL
116     RsaPrivateKeyOp(
117         bigNum               inOut, // IN/OUT: number to be exponentiated
118         privateExponent     *Z
119         )
120     {
121         BN_RSA(M1);
122         BN_RSA(M2);
123         BN_RSA(M);
124         BN_RSA(H);
125     //
126         MakePgreaterThanQ(Z);
127         // m1 = cdP mod p
128         VERIFY(BnModExp(M1, inOut, Z->dP, Z->P));
129         // m2 = cdQ mod q
130         VERIFY(BnModExp(M2, inOut, Z->dQ, Z->Q));
131         // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
132         // so m2 < P
133         VERIFY(BnSub(H, Z->P, M2));
134         VERIFY(BnAdd(H, H, M1));
135         VERIFY(BnModMult(H, H, Z->qInv, Z->P));
136         // m = m2 + h * q
137         VERIFY(BnMult(M, H, Z->Q));
138         VERIFY(BnAdd(inOut, M2, M));
139         return TRUE;
140     Error:
141         return FALSE;
142     }
```

10.2.17.4.7 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

```
143     static TPM_RC
144     RSAEP(
145         TPM2B       *dInOut,        // IN: size of the encrypted block and the size of
146                                     //     the encrypted value. It must be the size of
```

```
147                                 //     the modulus.
148                                 // OUT: the encrypted data. Will receive the
149                                 //      decrypted value
150     OBJECT      *key            // IN: the key to use
151     )
152 {
153     TPM2B_TYPE(4BYTES, 4);
154     TPM2B_4BYTES        e2B;
155     UINT32              e = key->publicArea.parameters.rsaDetail.exponent;
156 //
157     if(e == 0)
158         e = RSA_DEFAULT_PUBLIC_EXPONENT;
159     UINT32_TO_BYTE_ARRAY(e, e2B.t.buffer);
160     e2B.t.size = 4;
161     return ModExpB(dInOut->size, dInOut->buffer, dInOut->size, dInOut->buffer,
162                    e2B.t.size, e2B.t.buffer, key->publicArea.unique.rsa.t.size,
163                    key->publicArea.unique.rsa.t.buffer);
164 }
```

10.2.17.4.8 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

```
165 static TPM_RC
166 RSADP(
167     TPM2B           *inOut,         // IN/OUT: the value to encrypt
168     OBJECT          *key            // IN: the key
169     )
170 {
171     BN_RSA_INITIALIZED(bnM, inOut);
172     NEW_PRIVATE_EXPONENT(Z);
173     if(UnsignedCompareB(inOut->size, inOut->buffer,
174                         key->publicArea.unique.rsa.t.size,
175                         key->publicArea.unique.rsa.t.buffer) >= 0)
176         return TPM_RC_SIZE;
177     // private key operation requires that private exponent be loaded
178     // During self-test, this might not be the case so load it up if it hasn't
179     // already done
180     // been done
181     if((key->sensitive.sensitive.rsa.t.size & RSA_prime_flag) == 0)
182     {
183         if(CryptRsaLoadPrivateExponent(&key->publicArea, &key->sensitive)
184            != TPM_RC_SUCCESS)
185             return TPM_RC_BINDING;
186     }
187     VERIFY(UnpackExponent(&key->sensitive.sensitive.rsa, Z));
188     VERIFY(RsaPrivateKeyOp(bnM, Z));
189     VERIFY(BnTo2B(bnM, inOut, inOut->size));
190     return TPM_RC_SUCCESS;
191 Error:
192     return TPM_RC_FAILURE;
193 }
```

10.2.17.4.9 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

```
194  static TPM_RC
195  OaepEncode(
196      TPM2B       *padded,        // OUT: the pad data
197      TPM_ALG_ID   hashAlg,       // IN: algorithm to use for padding
198      const TPM2B *label,         // IN: null-terminated string (may be NULL)
199      TPM2B       *message,       // IN: the message being padded
200      RAND_STATE  *rand           // IN: the random number generator to use
201      )
202  {
203      INT32         padLen;
204      INT32         dbSize;
205      INT32         i;
206      BYTE          mySeed[MAX_DIGEST_SIZE];
207      BYTE         *seed = mySeed;
208      UINT16        hLen = CryptHashGetDigestSize(hashAlg);
209      BYTE          mask[MAX_RSA_KEY_BYTES];
210      BYTE         *pp;
211      BYTE         *pm;
212      TPM_RC        retVal = TPM_RC_SUCCESS;
213
214      pAssert(padded != NULL && message != NULL);
215
216      // A value of zero is not allowed because the KDF can't produce a result
217      // if the digest size is zero.
218      if(hLen == 0)
219          return TPM_RC_VALUE;
220
221      // Basic size checks
222      //  make sure digest isn't too big for key size
223      if(padded->size < (2 * hLen) + 2)
224          ERROR_RETURN(TPM_RC_HASH);
225
226      // and that message will fit messageSize <= k - 2hLen - 2
227      if(message->size > (padded->size - (2 * hLen) - 2))
228          ERROR_RETURN(TPM_RC_VALUE);
229
230      // Hash L even if it is null
231      // Offset into padded leaving room for masked seed and byte of zero
232      pp = &padded->buffer[hLen + 1];
233      if(CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
234                        hLen, pp) != hLen)
235          ERROR_RETURN(TPM_RC_FAILURE);
236
237      // concatenate PS of k  mLen  2hLen  2
238      padLen = padded->size - message->size - (2 * hLen) - 2;
239      MemorySet(&pp[hLen], 0, padLen);
240      pp[hLen + padLen] = 0x01;
241      padLen += 1;
242      memcpy(&pp[hLen + padLen], message->buffer, message->size);
243
244      // The total size of db = hLen + pad + mSize;
245      dbSize = hLen + padLen + message->size;
246
247      // If testing, then use the provided seed. Otherwise, use values
248      // from the RNG
249      CryptRandomGenerate(hLen, mySeed);
```

```
250         DRBG_Generate(rand, mySeed, (UINT16)hLen);
251         if(g_inFailureMode)
252             ERROR_RETURN(TPM_RC_FAILURE);
253         // mask = MGF1 (seed, nSize  hLen  1)
254         CryptMGF1(dbSize, mask, hashAlg, hLen, seed);
255
256         // Create the masked db
257         pm = mask;
258         for(i = dbSize; i > 0; i--)
259             *pp++ ^= *pm++;
260         pp = &padded->buffer[hLen + 1];
261
262         // Run the masked data through MGF1
263         if(CryptMGF1(hLen, &padded->buffer[1], hashAlg, dbSize, pp) != (unsigned)hLen)
264             ERROR_RETURN(TPM_RC_VALUE);
265     // Now XOR the seed to create masked seed
266         pp = &padded->buffer[1];
267         pm = seed;
268         for(i = hLen; i > 0; i--)
269             *pp++ ^= *pm++;
270         // Set the first byte to zero
271         padded->buffer[0] = 0x00;
272     Exit:
273         return retVal;
274     }
```

10.2.17.4.10 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

```
275     static TPM_RC
276     OaepDecode(
277         TPM2B           *dataOut,       // OUT: the recovered data
278         TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
279         const TPM2B     *label,         // IN: null-terminated string (may be NULL)
280         TPM2B           *padded         // IN: the padded data
281         )
282     {
283         UINT32       i;
284         BYTE         seedMask[MAX_DIGEST_SIZE];
285         UINT32       hLen = CryptHashGetDigestSize(hashAlg);
286
287         BYTE         mask[MAX_RSA_KEY_BYTES];
288         BYTE        *pp;
289         BYTE        *pm;
290         TPM_RC       retVal = TPM_RC_SUCCESS;
291
292         // Strange size (anything smaller can't be an OAEP padded block)
293         // Also check for no leading 0
294         if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
295             ERROR_RETURN(TPM_RC_VALUE);
296     // Use the hash size to determine what to put through MGF1 in order
297     // to recover the seedMask
298         CryptMGF1(hLen, seedMask, hashAlg, padded->size - hLen - 1,
299                   &padded->buffer[hLen + 1]);
```

```
300
301         // Recover the seed into seedMask
302         pAssert(hLen <= sizeof(seedMask));
303         pp = &padded->buffer[1];
304         pm = seedMask;
305         for(i = hLen; i > 0; i--)
306             *pm++ ^= *pp++;
307
308         // Use the seed to generate the data mask
309         CryptMGF1(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask);
310
311         // Use the mask generated from seed to recover the padded data
312         pp = &padded->buffer[hLen + 1];
313         pm = mask;
314         for(i = (padded->size - hLen - 1); i > 0; i--)
315             *pm++ ^= *pp++;
316
317         // Make sure that the recovered data has the hash of the label
318         // Put trial value in the seed mask
319         if((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
320                            hLen, seedMask)) != hLen)
321             FAIL(FATAL_ERROR_INTERNAL);
322         if(memcmp(seedMask, mask, hLen) != 0)
323             ERROR_RETURN(TPM_RC_VALUE);
324
325         // find the start of the data
326         pm = &mask[hLen];
327         for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
328         {
329             if(*pm++ != 0)
330                 break;
331         }
332         // If we ran out of data or didn't end with 0x01, then return an error
333         if(i == 0 || pm[-1] != 0x01)
334             ERROR_RETURN(TPM_RC_VALUE);
335
336         // pm should be pointing at the first part of the data
337         // and i is one greater than the number of bytes to move
338         i--;
339         if(i > dataOut->size)
340             // Special exit to preserve the size of the output buffer
341             return TPM_RC_VALUE;
342         memcpy(dataOut->buffer, pm, i);
343         dataOut->size = (UINT16)i;
344     Exit:
345         if(retVal != TPM_RC_SUCCESS)
346             dataOut->size = 0;
347         return retVal;
348     }
```

10.2.17.4.11 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

```
349     static TPM_RC
350     RSAES_PKCS1v1_5Encode(
351         TPM2B       *padded,        // OUT: the pad data
352         TPM2B       *message,       // IN: the message being padded
353         RAND_STATE  *rand
354         )
```

```
355 {
356     UINT32      ps = padded->size - message->size - 3;
357 //
358     if(message->size > padded->size - 11)
359         return TPM_RC_VALUE;
360     // move the message to the end of the buffer
361     memcpy(&padded->buffer[padded->size - message->size], message->buffer,
362            message->size);
363     // Set the first byte to 0x00 and the second to 0x02
364     padded->buffer[0] = 0;
365     padded->buffer[1] = 2;
366
367     // Fill with random bytes
368     DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
369     if(g_inFailureMode)
370         return TPM_RC_FAILURE;
371
372     // Set the delimiter for the random field to 0
373     padded->buffer[2 + ps] = 0;
374
375     // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
376     // In this implementation, use the value of the current index
377     for(ps++; ps > 1; ps--)
378     {
379         if(padded->buffer[ps] == 0)
380             padded->buffer[ps] = 0x55;  // In the < 0.5% of the cases that the
381                                         // random value is 0, just pick a value to
382                                         // put into the spot.
383     }
384     return TPM_RC_SUCCESS;
385 }
```

10.2.17.4.12 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

```
386 static TPM_RC
387 RSAES_Decode(
388     TPM2B       *message,       // OUT: the recovered message
389     TPM2B       *coded          // IN: the encoded message
390     )
391 {
392     BOOL        fail = FALSE;
393     UINT16      pSize;
394
395     fail = (coded->size < 11);
396     fail = (coded->buffer[0] != 0x00) | fail;
397     fail = (coded->buffer[1] != 0x02) | fail;
398     for(pSize = 2; pSize < coded->size; pSize++)
399     {
400         if(coded->buffer[pSize] == 0)
401             break;
402     }
403     pSize++;
404
405     // Make sure that pSize has not gone over the end and that there are at least 8
406     // bytes of pad data.
407     fail = (pSize > coded->size) | fail;
408     fail = ((pSize - 2) < 8) | fail;
409     if((message->size < (UINT16)(coded->size - pSize)) || fail)
```

```
410             return TPM_RC_VALUE;
411         message->size = coded->size - pSize;
412         memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
413         return TPM_RC_SUCCESS;
414     }
```

10.2.17.4.13 CryptRsaPssSaltSize()

This function computes the salt size used in PSS. It is broken out so that the X509 code can get the same value that is used by the encoding function in this module.

```
415     INT16
416     CryptRsaPssSaltSize(
417         INT16               hashSize,
418         INT16                outSize
419     )
420     {
421         INT16                saltSize;
422     //
423         // (Mask Length) = (outSize - hashSize - 1);
424         // Max saltSize is (Mask Length) - 1
425         saltSize = (outSize - hashSize - 1) - 1;
426         // Use the maximum salt size allowed by FIPS 186-4
427         if(saltSize > hashSize)
428             saltSize = hashSize;
429         else if(saltSize < 0)
430             saltSize = 0;
431         return saltSize;
432     }
```

10.2.17.4.14 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.

```
433     static TPM_RC
434     PssEncode(
435         TPM2B           *out,        // OUT: the encoded buffer
436         TPM_ALG_ID       hashAlg,    // IN: hash algorithm for the encoding
437         TPM2B           *digest,     // IN: the digest
438         RAND_STATE      *rand        // IN: random number source
439         )
440     {
441         UINT32               hLen = CryptHashGetDigestSize(hashAlg);
442         BYTE                 salt[MAX_RSA_KEY_BYTES - 1];
443         UINT16               saltSize;
444         BYTE                *ps = salt;
445         BYTE                *pOut;
446         UINT16               mLen;
447         HASH_STATE           hashState;
448
449         // These are fatal errors indicating bad TPM firmware
450         pAssert(out != NULL && hLen > 0 && digest != NULL);
451
452         // Get the size of the mask
453         mLen = (UINT16)(out->size - hLen - 1);
454
455         // Set the salt size
456         saltSize = CryptRsaPssSaltSize((INT16)hLen, (INT16)out->size);
457
458     //using eOut for scratch space
```

```
459         // Set the first 8 bytes to zero
460         pOut = out->buffer;
461         memset(pOut, 0, 8);
462 
463         // Get set the salt
464         DRBG_Generate(rand, salt, saltSize);
465         if(g_inFailureMode)
466             return TPM_RC_FAILURE;
467 
468         // Create the hash of the pad || input hash || salt
469         CryptHashStart(&hashState, hashAlg);
470         CryptDigestUpdate(&hashState, 8, pOut);
471         CryptDigestUpdate2B(&hashState, digest);
472         CryptDigestUpdate(&hashState, saltSize, salt);
473         CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
474 
475         // Create a mask
476         if(CryptMGF1(mLen, pOut, hashAlg, hLen, &pOut[mLen]) != mLen)
477             FAIL(FATAL_ERROR_INTERNAL);
478 
479         // Since this implementation uses key sizes that are all even multiples of
480         // 8, just need to make sure that the most significant bit is CLEAR
481         *pOut &= 0x7f;
482 
483         // Before we mess up the pOut value, set the last byte to 0xbc
484         pOut[out->size - 1] = 0xbc;
485 
486         // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
487         pOut = &pOut[mLen - saltSize - 1];
488         *pOut++ ^= 0x01;
489 
490         // XOR the salt data into the buffer
491         for(; saltSize > 0; saltSize--)
492             *pOut++ ^= *ps++;
493 
494         // and we are done
495         return TPM_RC_SUCCESS;
496     }
```

10.2.17.4.15 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

```
497     static TPM_RC
498     PssDecode(
499         TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
500         TPM2B       *dIn,            // In: the digest to compare
501         TPM2B       *eIn             // IN: the encoded data
502         )
503     {
504         UINT32           hLen = CryptHashGetDigestSize(hashAlg);
505         BYTE             mask[MAX_RSA_KEY_BYTES];
```

```
506     BYTE            *pm = mask;
507     BYTE            *pe;
508     BYTE             pad[8] = {0};
509     UINT32           i;
510     UINT32           mLen;
511     BYTE             fail;
512     TPM_RC           retVal = TPM_RC_SUCCESS;
513     HASH_STATE       hashState;
514
515     // These errors are indicative of failures due to programmer error
516     pAssert(dIn != NULL && eIn != NULL);
517     pe = eIn->buffer;
518
519     // check the hash scheme
520     if(hLen == 0)
521         ERROR_RETURN(TPM_RC_SCHEME);
522
523     // most significant bit must be zero
524     fail = pe[0] & 0x80;
525
526     // last byte must be 0xbc
527     fail |= pe[eIn->size - 1] ^ 0xbc;
528
529     // Use the hLen bytes at the end of the buffer to generate a mask
530     // Doesn't start at the end which is a flag byte
531     mLen = eIn->size - hLen - 1;
532     CryptMGF1(mLen, mask, hashAlg, hLen, &pe[mLen]);
533
534     // Clear the MSO of the mask to make it consistent with the encoding.
535     mask[0] &= 0x7F;
536
537     pAssert(mLen <= sizeof(mask));
538     // XOR the data into the mask to recover the salt. This sequence
539     // advances eIn so that it will end up pointing to the seed data
540     // which is the hash of the signature data
541     for(i = mLen; i > 0; i--)
542         *pm++ ^= *pe++;
543
544     // Find the first byte of 0x01 after a string of all 0x00
545     for(pm = mask, i = mLen; i > 0; i--)
546     {
547         if(*pm == 0x01)
548             break;
549         else
550             fail |= *pm++;
551     }
552     // i should not be zero
553     fail |= (i == 0);
554
555     // if we have failed, will continue using the entire mask as the salt value so
556     // that the timing attacks will not disclose anything (I don't think that this
557     // is a problem for TPM applications but, usually, we don't fail so this
558     // doesn't cost anything).
559     if(fail)
560     {
561         i = mLen;
562         pm = mask;
563     }
564     else
565     {
566         pm++;
567         i--;
568     }
569     // i contains the salt size and pm points to the salt. Going to use the input
570     // hash and the seed to recreate the hash in the lower portion of eIn.
571     CryptHashStart(&hashState, hashAlg);
```

```
572
573         // add the pad of 8 zeros
574         CryptDigestUpdate(&hashState, 8, pad);
575
576         // add the provided digest value
577         CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
578
579         // and the salt
580         CryptDigestUpdate(&hashState, i, pm);
581
582         // get the result
583         fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
584
585         // Compare all bytes
586         for(pm = mask; hLen > 0; hLen--)
587             // don't use fail = because that could skip the increment and compare
588             // operations after the first failure and that gives away timing
589             // information.
590             fail |= *pm++ ^ *pe++;
591
592         retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
593     Exit:
594             return retVal;
595     }
```

10.2.17.4.16 MakeDerTag()

Construct the DER value that is used in RSASSA

Return Value	Meaning
0	size of value
0	no hash exists

```
596     INT16
597     MakeDerTag(
598         TPM_ALG_ID   hashAlg,
599         INT16        sizeOfBuffer,
600         BYTE        *buffer
601     )
602     {
603     //    0x30, 0x31,       // SEQUENCE (2 elements) 1st
604     //        0x30, 0x0D,   // SEQUENCE (2 elements)
605     //            0x06, 0x09,   // HASH OID
606     //                0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
607     //             0x05, 0x00,  // NULL
608     //        0x04, 0x20  //  OCTET STRING
609         HASH_DEF   *info = CryptGetHashDef(hashAlg);
610         INT16       oidSize;
611         // If no OID, can't do encode
612         VERIFY(info != NULL);
613         oidSize = 2 + (info->OID)[1];
614         // make sure this fits in the buffer
615         VERIFY(sizeOfBuffer >= (oidSize + 8));
616         *buffer++ = 0x30;  // 1st SEQUENCE
617         // Size of the 1st SEQUENCE is 6 bytes + size of the hash OID + size of the
618         // digest size
619         *buffer++ = (BYTE)(6 + oidSize + info->digestSize);   //
620         *buffer++ = 0x30; // 2nd SEQUENCE
621         // size is 4 bytes of overhead plus the side of the OID
622         *buffer++ = (BYTE)(2 + oidSize);
623         MemoryCopy(buffer, info->OID, oidSize);
624         buffer += oidSize;
```

```
625         *buffer++ = 0x05;    // Add a NULL
626         *buffer++ = 0x00;
627
628         *buffer++ = 0x04;
629         *buffer++ = (BYTE)(info->digestSize);
630         return oidSize + 8;
631     Error:
632         return 0;
633
634     }
```

10.2.17.4.17 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*

```
635     static TPM_RC
636     RSASSA_Encode(
637         TPM2B               *pOut,      // IN:OUT on in, the size of the public key
638                                         //        on out, the encoded area
639         TPM_ALG_ID           hashAlg,   // IN: hash algorithm for PKCS1v1_5
640         TPM2B               *hIn        // IN: digest value to encode
641         )
642     {
643         BYTE             DER[20];
644         BYTE            *der = DER;
645         INT32            derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
646         BYTE            *eOut;
647         INT32            fillSize;
648         TPM_RC           retVal = TPM_RC_SUCCESS;
649
650         // Can't use this scheme if the algorithm doesn't have a DER string defined.
651         if(derSize == 0)
652             ERROR_RETURN(TPM_RC_SCHEME);
653
654         // If the digest size of 'hashAl' doesn't match the input digest size, then
655         // the DER will misidentify the digest so return an error
656         if(CryptHashGetDigestSize(hashAlg) != hIn->size)
657             ERROR_RETURN(TPM_RC_VALUE);
658         fillSize = pOut->size - derSize - hIn->size - 3;
659         eOut = pOut->buffer;
660
661         // Make sure that this combination will fit in the provided space
662         if(fillSize < 8)
663             ERROR_RETURN(TPM_RC_SIZE);
664
665         // Start filling
666         *eOut++ = 0; // initial byte of zero
667         *eOut++ = 1; // byte of 0x01
668         for(; fillSize > 0; fillSize--)
669             *eOut++ = 0xff; // bunch of 0xff
670         *eOut++ = 0; // another 0
671         for(; derSize > 0; derSize--)
672             *eOut++ = *der++;    // copy the DER
673         der = hIn->buffer;
674         for(fillSize = hIn->size; fillSize > 0; fillSize--)
675             *eOut++ = *der++;    // copy the hash
676     Exit:
```

```
677         return retVal;
678     }
```

10.2.17.4.18 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

```
679     static TPM_RC
680     RSASSA_Decode(
681         TPM_ALG_ID       hashAlg,        // IN: hash algorithm to use for the encoding
682         TPM2B           *hIn,            // In: the digest to compare
683         TPM2B           *eIn             // IN: the encoded data
684         )
685     {
686         BYTE             fail;
687         BYTE             DER[20];
688         BYTE            *der = DER;
689         INT32            derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
690         BYTE            *pe;
691         INT32            hashSize = CryptHashGetDigestSize(hashAlg);
692         INT32            fillSize;
693         TPM_RC           retVal;
694         BYTE            *digest;
695         UINT16           digestSize;
696
697         pAssert(hIn != NULL && eIn != NULL);
698         pe = eIn->buffer;
699
700         // Can't use this scheme if the algorithm doesn't have a DER string
701         // defined or if the provided hash isn't the right size
702         if(derSize == 0 || (unsigned)hashSize != hIn->size)
703             ERROR_RETURN(TPM_RC_SCHEME);
704
705         // Make sure that this combination will fit in the provided space
706         // Since no data movement takes place, can just walk though this
707         // and accept nearly random values. This can only be called from
708         // CryptValidateSignature() so eInSize is known to be in range.
709         fillSize = eIn->size - derSize - hashSize - 3;
710
711         // Start checking (fail will become non-zero if any of the bytes do not have
712         // the expected value.
713         fail = *pe++;                   // initial byte of zero
714         fail |= *pe++ ^ 1;              // byte of 0x01
715         for(; fillSize > 0; fillSize--)
716             fail |= *pe++ ^ 0xff;       // bunch of 0xff
717         fail |= *pe++;                  // another 0
718         for(; derSize > 0; derSize--)
719             fail |= *pe++ ^ *der++;    // match the DER
720         digestSize = hIn->size;
721         digest = hIn->buffer;
722         for(; digestSize > 0; digestSize--)
723             fail |= *pe++ ^ *digest++; // match the hash
724         retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
725     Exit:
726         return retVal;
727     }
```

10.2.17.5 Externally Accessible Functions

10.2.17.5.1 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.

```
728 TPMT_RSA_DECRYPT*
729 CryptRsaSelectScheme(
730     TPMI_DH_OBJECT       rsaHandle,     // IN: handle of an RSA key
731     TPMT_RSA_DECRYPT    *scheme         // IN: a sign or decrypt scheme
732     )
733 {
734     OBJECT              *rsaObject;
735     TPMT_ASYM_SCHEME    *keyScheme;
736     TPMT_RSA_DECRYPT    *retVal = NULL;
737
738     // Get sign object pointer
739     rsaObject = HandleToObject(rsaHandle);
740     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
741
742     // if the default scheme of the object is TPM_ALG_NULL, then select the
743     // input scheme
744     if(keyScheme->scheme == TPM_ALG_NULL)
745     {
746         retVal = scheme;
747     }
748     // if the object scheme is not TPM_ALG_NULL and the input scheme is
749     // TPM_ALG_NULL, then select the default scheme of the object.
750     else if(scheme->scheme == TPM_ALG_NULL)
751     {
752         // if input scheme is NULL
753         retVal = (TPMT_RSA_DECRYPT *)keyScheme;
754     }
755     // get here if both the object scheme and the input scheme are
756     // not TPM_ALG_NULL. Need to insure that they are the same.
757     // IMPLEMENTATION NOTE: This could cause problems if future versions have
758     // schemes that have more values than just a hash algorithm. A new function
759     // (IsSchemeSame()) might be needed then.
760     else if(keyScheme->scheme == scheme->scheme
761             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
762     {
763         retVal = scheme;
764     }
765     // two different, incompatible schemes specified will return NULL
766     return retVal;
767 }
```

10.2.17.5.2 CryptRsaLoadPrivateExponent()

This function is called to generate the private exponent of an RSA key.

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
768 TPM_RC
769 CryptRsaLoadPrivateExponent(
770     TPMT_PUBLIC             *publicArea,
771     TPMT_SENSITIVE          *sensitive
772     )
773 {
774 //
775     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) == 0)
776     {
777         if((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size)
778         {
779             NEW_PRIVATE_EXPONENT(Z);
780             BN_RSA_INITIALIZED(bnN, &publicArea->unique.rsa);
781             BN_RSA(bnQr);
782             BN_VAR(bnE, RADIX_BITS);
783 
784             TEST(ALG_NULL_VALUE);
785 
786             VERIFY((sensitive->sensitive.rsa.t.size * 2)
787                    == publicArea->unique.rsa.t.size);
788             // Initialize the exponent
789             BnSetWord(bnE, publicArea->parameters.rsaDetail.exponent);
790             if(BnEqualZero(bnE))
791                 BnSetWord(bnE, RSA_DEFAULT_PUBLIC_EXPONENT);
792             // Convert first prime to 2B
793             VERIFY(BnFrom2B(Z->P, &sensitive->sensitive.rsa.b) != NULL);
794 
795             // Find the second prime by division. This uses 'bQ' rather than Z->Q
796             // because the division could make the quotient larger than a prime during
797             // some intermediate step.
798             VERIFY(BnDiv(Z->Q, bnQr, bnN, Z->P));
799             VERIFY(BnEqualZero(bnQr));
800             // Compute the private exponent and return it if found
801             VERIFY(ComputePrivateExponent(bnE, Z));
802             VERIFY(PackExponent(&sensitive->sensitive.rsa, Z));
803         }
804         else
805             VERIFY(((sensitive->sensitive.rsa.t.size / 5) * 2)
806                    == publicArea->unique.rsa.t.size);
807         sensitive->sensitive.rsa.t.size |= RSA_prime_flag;
808     }
809     return TPM_RC_SUCCESS;
810 Error:
811     return TPM_RC_BINDING;
812 }
```

10.2.17.5.3 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

```
813  LIB_EXPORT TPM_RC
814  CryptRsaEncrypt(
815      TPM2B_PUBLIC_KEY_RSA        *cOut,          // OUT: the encrypted data
816      TPM2B                       *dIn,           // IN: the data to encrypt
817      OBJECT                      *key,           // IN: the key used for encryption
818      TPMT_RSA_DECRYPT            *scheme,        // IN: the type of padding and hash
819                                                  //     if needed
820      const TPM2B                 *label,         // IN: in case it is needed
821      RAND_STATE                  *rand           // IN: random number generator
822                                                  //     state (mostly for testing)
823      )
824  {
825      TPM_RC                       retVal = TPM_RC_SUCCESS;
826      TPM2B_PUBLIC_KEY_RSA         dataIn;
827  //
828      // if the input and output buffers are the same, copy the input to a scratch
829      // buffer so that things don't get messed up.
830      if(dIn == &cOut->b)
831      {
832          MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
833          dIn = &dataIn.b;
834      }
835      // All encryption schemes return the same size of data
836      cOut->t.size = key->publicArea.unique.rsa.t.size;
837      TEST(scheme->scheme);
838
839      switch(scheme->scheme)
840      {
841          case ALG_NULL_VALUE:  // 'raw' encryption
842          {
843              INT32            i;
844              INT32            dSize = dIn->size;
845              // dIn can have more bytes than cOut as long as the extra bytes
846              // are zero. Note: the more significant bytes of a number in a byte
847              // buffer are the bytes at the start of the array.
848              for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
849              dSize -= i;
850              if(dSize > cOut->t.size)
851                  ERROR_RETURN(TPM_RC_VALUE);
852              // Pad cOut with zeros if dIn is smaller
853              memset(cOut->t.buffer, 0, cOut->t.size - dSize);
854              // And copy the rest of the value
855              memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
856
857              // If the size of dIn is the same as cOut dIn could be larger than
858              // the modulus. If it is, then RSAEP() will catch it.
859          }
860          break;
861          case ALG_RSAES_VALUE:
862              retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
863              break;
864          case ALG_OAEP_VALUE:
865              retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
866                                  rand);
```

```
867             break;
868         default:
869             ERROR_RETURN(TPM_RC_SCHEME);
870             break;
871     }
872     // All the schemes that do padding will come here for the encryption step
873     // Check that the Encoding worked
874     if(retVal == TPM_RC_SUCCESS)
875         // Padding OK so do the encryption
876         retVal = RSAEP(&cOut->b, key);
877 Exit:
878     return retVal;
879 }
```

10.2.17.5.4 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

```
880 LIB_EXPORT TPM_RC
881 CryptRsaDecrypt(
882     TPM2B               *dOut,          // OUT: the decrypted data
883     TPM2B               *cIn,           // IN: the data to decrypt
884     OBJECT              *key,           // IN: the key to use for decryption
885     TPMT_RSA_DECRYPT    *scheme,        // IN: the padding scheme
886     const TPM2B         *label          // IN: in case it is needed for the scheme
887     )
888 {
889     TPM_RC                  retVal;
890
891     // Make sure that the necessary parameters are provided
892     pAssert(cIn != NULL && dOut != NULL && key != NULL);
893
894     // Size is checked to make sure that the encrypted value is the right size
895     if(cIn->size != key->publicArea.unique.rsa.t.size)
896         ERROR_RETURN(TPM_RC_SIZE);
897
898     TEST(scheme->scheme);
899
900     // For others that do padding, do the decryption in place and then
901     // go handle the decoding.
902     retVal = RSADP(cIn, key);
903     if(retVal == TPM_RC_SUCCESS)
904     {
905         // Remove padding
906         switch(scheme->scheme)
907         {
908             case ALG_NULL_VALUE:
909                 if(dOut->size < cIn->size)
910                     return TPM_RC_VALUE;
911                 MemoryCopy2B(dOut, cIn, dOut->size);
912                 break;
913             case ALG_RSAES_VALUE:
914                 retVal = RSAES_Decode(dOut, cIn);
915                 break;
916             case ALG_OAEP_VALUE:
```

```
                retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);
                break;
            default:
                retVal = TPM_RC_SCHEME;
                break;
        }
    }
Exit:
    return retVal;
}
```

10.2.17.5.5 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)

```
LIB_EXPORT TPM_RC
CryptRsaSign(
    TPMT_SIGNATURE      *sigOut,
    OBJECT              *key,           // IN: key to use
    TPM2B_DIGEST        *hIn,           // IN: the digest to sign
    RAND_STATE          *rand           // IN: the random number generator
                                        //      to use (mostly for testing)
    )
{
    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);
    }
```

```
970         return retVal;
971     }
```

10.2.17.5.6 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

```
972     LIB_EXPORT TPM_RC
973     CryptRsaValidateSignature(
974         TPMT_SIGNATURE  *sig,           // IN: signature
975         OBJECT          *key,           // IN: public modulus
976         TPM2B_DIGEST    *digest         // IN: The digest being validated
977         )
978     {
979         TPM_RC          retVal;
980     //
981         // Fatal programming errors
982         pAssert(key != NULL && sig != NULL && digest != NULL);
983         switch(sig->sigAlg)
984         {
985             case ALG_RSAPSS_VALUE:
986             case ALG_RSASSA_VALUE:
987                 break;
988             default:
989                 return TPM_RC_SCHEME;
990         }
991
992         // Errors that might be caused by calling parameters
993         if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
994             ERROR_RETURN(TPM_RC_SIGNATURE);
995
996         TEST(sig->sigAlg);
997
998         // Decrypt the block
999         retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
1000        if(retVal == TPM_RC_SUCCESS)
1001        {
1002            switch(sig->sigAlg)
1003            {
1004                case ALG_RSAPSS_VALUE:
1005                    retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
1006                                       &sig->signature.rsassa.sig.b);
1007                    break;
1008                case ALG_RSASSA_VALUE:
1009                    retVal = RSASSA_Decode(sig->signature.any.hashAlg, &digest->b,
1010                                           &sig->signature.rsassa.sig.b);
1011                    break;
1012                default:
1013                    return TPM_RC_SCHEME;
1014            }
1015        }
1016    Exit:
1017        return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
1018    }
1019    #if SIMULATION && USE_RSA_KEY_CACHE
1020    extern int s_rsaKeyCacheEnabled;
```

```
1021    int GetCachedRsaKey(TPMT_PUBLIC *publicArea, TPMT_SENSITIVE *sensitive,
1022                        RAND_STATE *rand);
1023    #define GET_CACHED_KEY(publicArea, sensitive, rand)                        \
1024                (s_rsaKeyCacheEnabled && GetCachedRsaKey(publicArea, sensitive, rand))
1025    #else
1026    #define GET_CACHED_KEY(key, rand)
1027    #endif
```

10.2.17.5.7 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

```
1028    LIB_EXPORT TPM_RC
1029    CryptRsaGenerateKey(
1030        TPMT_PUBLIC         *publicArea,
1031        TPMT_SENSITIVE      *sensitive,
1032        RAND_STATE          *rand               // IN: if not NULL, the deterministic
1033                                                //     RNG state
1034        )
1035    {
1036        UINT32               i;
1037        BN_RSA(bnD);
1038        BN_RSA(bnN);
1039        BN_WORD(bnPubExp);
1040        UINT32               e = publicArea->parameters.rsaDetail.exponent;
1041        int                  keySizeInBits;
1042        TPM_RC               retVal = TPM_RC_NO_RESULT;
1043        NEW_PRIVATE_EXPONENT(Z);
1044    //
1045
1046    // Need to make sure that the caller did not specify an exponent that is
1047    // not supported
1048        e = publicArea->parameters.rsaDetail.exponent;
1049        if(e == 0)
1050            e = RSA_DEFAULT_PUBLIC_EXPONENT;
1051        else
1052        {
1053            if(e < 65537)
1054                ERROR_RETURN(TPM_RC_RANGE);
1055            // Check that e is prime
1056            if(!IsPrimeInt(e))
1057                ERROR_RETURN(TPM_RC_RANGE);
1058        }
1059        BnSetWord(bnPubExp, e);
1060
1061        // check for supported key size.
1062        keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
1063        if(((keySizeInBits % 1024) != 0)
1064           || (keySizeInBits > MAX_RSA_KEY_BITS)  // this might be redundant, but...
1065           || (keySizeInBits == 0))
1066            ERROR_RETURN(TPM_RC_VALUE);
1067
1068        // Set the prime size for instrumentation purposes
1069        INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
1070
1071    #if SIMULATION && USE_RSA_KEY_CACHE
1072        if(GET_CACHED_KEY(publicArea, sensitive, rand))
```

```
1073            return TPM_RC_SUCCESS;
1074    #endif
1075
1076        // Make sure that key generation has been tested
1077        TEST(ALG_NULL_VALUE);
1078
1079        // The prime is computed in P. When a new prime is found, Q is checked to
1080        // see if it is zero.  If so, P is copied to Q and a new P is found.
1081        // When both P and Q are non-zero, the modulus and
1082        // private exponent are computed and a trial encryption/decryption is
1083        // performed.  If the encrypt/decrypt fails, assume that at least one of the
1084        // primes is composite. Since we don't know which one, set Q to zero and start
1085        // over and find a new pair of primes.
1086
1087        for(i = 1; (retVal == TPM_RC_NO_RESULT) && (i != 100); i++)
1088        {
1089            if(_plat__IsCanceled())
1090                ERROR_RETURN(TPM_RC_CANCELED);
1091
1092            if(BnGeneratePrimeForRSA(Z->P, keySizeInBits / 2, e, rand) == TPM_RC_FAILURE)
1093            {
1094                retVal = TPM_RC_FAILURE;
1095                goto Exit;
1096            }
1097
1098            INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
1099
1100            // If this is the second prime, make sure that it differs from the
1101            // first prime by at least 2^100
1102            if(BnEqualZero(Z->Q))
1103            {
1104                // copy p to q and compute another prime in p
1105                BnCopy(Z->Q, Z->P);
1106                continue;
1107            }
1108            // Make sure that the difference is at least 100 bits. Need to do it this
1109            // way because the big numbers are only positive values
1110            if(BnUnsignedCmp(Z->P, Z->Q) < 0)
1111                BnSub(bnD, Z->Q, Z->P);
1112            else
1113                BnSub(bnD, Z->P, Z->Q);
1114            if(BnMsb(bnD) < 100)
1115                continue;
1116
1117            //Form the public modulus and set the unique value
1118            BnMult(bnN, Z->P, Z->Q);
1119            BnTo2B(bnN, &publicArea->unique.rsa.b,
1120                   (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
1121            // Make sure everything came out right. The MSb of the values must be one
1122            if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1123                || (publicArea->unique.rsa.t.size
1124                    != (NUMBYTES)BITS_TO_BYTES(keySizeInBits)))
1125                FAIL(FATAL_ERROR_INTERNAL);
1126
1127            // Make sure that we can form the private exponent values
1128            if(ComputePrivateExponent(bnPubExp, Z) != TRUE)
1129            {
1130                // If ComputePrivateExponent could not find an inverse for
1131                // Q, then copy P and recompute P. This might
1132                // cause both to be recomputed if P is also zero
1133                if(BnEqualZero(Z->Q))
1134                    BnCopy(Z->Q, Z->P);
1135                continue;
1136            }
1137
1138            // Pack the private exponent into the sensitive area
```

```
1139            PackExponent(&sensitive->sensitive.rsa, Z);
1140            // Make sure everything came out right. The MSb of the values must be one
1141            if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1142               || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
1143                FAIL(FATAL_ERROR_INTERNAL);
1144
1145            retVal = TPM_RC_SUCCESS;
1146            // Do a trial encryption decryption if this is a signing key
1147            if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
1148            {
1149                BN_RSA(temp1);
1150                BN_RSA(temp2);
1151                BnGenerateRandomInRange(temp1, bnN, rand);
1152
1153                // Encrypt with public exponent...
1154                BnModExp(temp2, temp1, bnPubExp, bnN);
1155                // ...  then decrypt with private exponent
1156                RsaPrivateKeyOp(temp2, Z);
1157
1158                // If the starting and ending values are not the same,
1159                // start over )-;
1160                if(BnUnsignedCmp(temp2, temp1) != 0)
1161                {
1162                    BnSetWord(Z->Q, 0);
1163                    retVal = TPM_RC_NO_RESULT;
1164                }
1165            }
1166        }
1167    Exit:
1168        return retVal;
1169    }
1170    #endif // ALG_RSA
```

10.2.18 CryptSmac.c

10.2.18.1 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.

10.2.18.2 Includes, Defines, and Typedefs

```
#define _CRYPT_HASH_C_
#include "Tpm.h"
#if SMAC_IMPLEMENTED
```

10.2.18.2.1 CryptSmacStart()

Function to start an SMAC.

```
UINT16
CryptSmacStart(
    HASH_STATE              *state,
    TPMU_PUBLIC_PARMS       *keyParameters,
    TPM_ALG_ID               macAlg,          // IN: the type of MAC
    TPM2B                   *key
)
{
    UINT16                  retVal = 0;
//
    // Make sure that the key size is correct. This should have been checked
    // at key load, but...
    if(BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
    {
        switch(macAlg)
        {
#if ALG_CMAC
            case ALG_CMAC_VALUE:
                retVal = CryptCmacStart(&state->state.smac, keyParameters,
                                        macAlg, key);
                break;
#endif
            default:
                break;
        }
    }
    state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
    return retVal;
}
```

10.2.18.2.2 CryptMacStart()

Function to start either an HMAC or an SMAC. Cannot reuse the CryptHmacStart() function because of the difference in number of parameters.

```
UINT16
CryptMacStart(
    HMAC_STATE              *state,
    TPMU_PUBLIC_PARMS       *keyParameters,
    TPM_ALG_ID               macAlg,          // IN: the type of MAC
    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  }
```

10.2.18.2.3 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  }
```

10.2.18.2.4 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
```

10.2.19 CryptSym.c

10.2.19.1 Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These functions only use the single block encryption functions of the selected symmetric crypto library.

10.2.19.2 Includes, Defines, and Typedefs

```
#include "Tpm.h"
#include "CryptSym.h"
#define     KEY_BLOCK_SIZES(ALG, alg)                                                   \
static const INT16       alg##KeyBlockSizes[] = {                                       \
                               ALG##_KEY_SIZES_BITS, -1, ALG##_BLOCK_SIZES };
#if ALG_AES
    KEY_BLOCK_SIZES(AES, aes);
#endif // ALG_AES
#if ALG_SM4
    KEY_BLOCK_SIZES(SM4, sm4);
#endif
#if ALG_CAMELLIA
    KEY_BLOCK_SIZES(CAMELLIA, camellia);
#endif
#if ALG_TDES
    KEY_BLOCK_SIZES(TDES, tdes);
#endif
```

10.2.19.3 Initialization and Data Access Functions

10.2.19.3.1 CryptSymInit()

This function is called to do _TPM_Init() processing

```
BOOL
CryptSymInit(
    void
    )
{
    return TRUE;
}
```

10.2.19.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```
BOOL
CryptSymStartup(
    void
    )
{
    return TRUE;
}
```

10.2.19.3.3 CryptGetSymmetricBlockSize()

This function returns the block size of the algorithm. The table of bit sizes has an entry for each allowed key size. The entry for a key size is 0 if the TPM does not implement that key size. The key size table is

delimited with a negative number (-1). After the delimiter is a list of block sizes with each entry corresponding to the key bit size. For most symmetric algorithms, the block size is the same regardless of the key size but this arrangement allows them to be different.

Return Value	Meaning
0	cipher not supported
0	the cipher block size in bytes

```
32  LIB_EXPORT INT16
33  CryptGetSymmetricBlockSize(
34      TPM_ALG_ID      symmetricAlg,   // IN: the symmetric algorithm
35      UINT16          keySizeInBits   // IN: the key size
36      )
37  {
38      const INT16    *sizes;
39      INT16           i;
40  #define ALG_CASE(SYM, sym)  case ALG_##SYM##_VALUE: sizes = sym##KeyBlockSizes; break
41      switch(symmetricAlg)
42      {
43  #if ALG_AES
44          ALG_CASE(AES, aes);
45  #endif
46  #if ALG_SM4
47          ALG_CASE(SM4, sm4);
48  #endif
49  #if ALG_CAMELLIA
50          ALG_CASE(CAMELLIA, camellia);
51  #endif
52  #if ALG_TDES
53          ALG_CASE(TDES, tdes);
54  #endif
55          default:
56              return 0;
57      }
58      // Find the index of the indicated keySizeInBits
59      for(i = 0; *sizes >= 0; i++, sizes++)
60      {
61          if(*sizes == keySizeInBits)
62              break;
63      }
64      // If sizes is pointing at the end of the list of key sizes, then the desired
65      // key size was not found so set the block size to zero.
66      if(*sizes++ < 0)
67          return 0;
68      // Advance until the end of the list is found
69      while(*sizes++ >= 0);
70      // sizes is pointing to the first entry in the list of block sizes. Use the
71      // ith index to find the block size for the corresponding key size.
72      return sizes[i];
73  }
```

10.2.19.4 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

```
74  LIB_EXPORT TPM_RC
75  CryptSymmetricEncrypt(
76      BYTE                *dOut,          // OUT:
77      TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
78      UINT16               keySizeInBits, // IN: key size in bits
79      const BYTE          *key,           // IN: key buffer. The size of this buffer
80                                          //     in bytes is (keySizeInBits + 7) / 8
81      TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
82      TPM_ALG_ID           mode,          // IN: Mode to use
83      INT32                dSize,         // IN: data size (may need to be a
84                                          //     multiple of the blockSize)
85      const BYTE          *dIn            // IN: data buffer
86      )
87  {
88      BYTE                *pIv;
89      int                  i;
90      BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
91      BYTE                *pT;
92      tpmCryptKeySchedule_t        keySchedule;
93      INT16                blockSize;
94      TpmCryptSetSymKeyCall_t      encrypt;
95      BYTE                *iv;
96      BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
97  //
98      pAssert(dOut != NULL && key != NULL && dIn != NULL);
99      if(dSize == 0)
100         return TPM_RC_SUCCESS;
101
102     TEST(algorithm);
103     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
104     if(blockSize == 0)
105         return TPM_RC_FAILURE;
106     // If the iv is provided, then it is expected to be block sized. In some cases,
107     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
108     // with no knowledge of the actual block size. This function will set it.
109     if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
110     {
111         ivInOut->t.size = blockSize;
112         iv = ivInOut->t.buffer;
113     }
114     else
115         iv = defaultIv;
116     pIv = iv;
117
118     // Create encrypt key schedule and set the encryption function pointer.
119
120     SELECT(ENCRYPT);
121
122     switch(mode)
123     {
124 #if ALG_CTR
125         case ALG_CTR_VALUE:
126             for(; dSize > 0; dSize -= blockSize)
127             {
128                 // Encrypt the current value of the IV(counter)
129                 ENCRYPT(&keySchedule, iv, tmp);
130
131                 //increment the counter (counter is big-endian so start at end)
```

```
132                 for(i = blockSize - 1; i >= 0; i--)
133                     if((iv[i] += 1) != 0)
134                         break;
135                 // XOR the encrypted counter value with input and put into output
136                 pT = tmp;
137                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
138                     *dOut++ = *dIn++ ^ *pT++;
139             }
140             break;
141 #endif
142 #if ALG_OFB
143         case ALG_OFB_VALUE:
144             // This is written so that dIn and dOut may be the same
145             for(; dSize > 0; dSize -= blockSize)
146             {
147                 // Encrypt the current value of the "IV"
148                 ENCRYPT(&keySchedule, iv, iv);
149 
150                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
151                 pIv = iv;
152                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
153                     *dOut++ = (*pIv++ ^ *dIn++);
154             }
155             break;
156 #endif
157 #if ALG_CBC
158         case ALG_CBC_VALUE:
159             // For CBC the data size must be an even multiple of the
160             // cipher block size
161             if((dSize % blockSize) != 0)
162                 return TPM_RC_SIZE;
163             // XOR the data block into the IV, encrypt the IV into the IV
164             // and then copy the IV to the output
165             for(; dSize > 0; dSize -= blockSize)
166             {
167                 pIv = iv;
168                 for(i = blockSize; i > 0; i--)
169                     *pIv++ ^= *dIn++;
170                 ENCRYPT(&keySchedule, iv, iv);
171                 pIv = iv;
172                 for(i = blockSize; i > 0; i--)
173                     *dOut++ = *pIv++;
174             }
175             break;
176 #endif
177         // CFB is not optional
178         case ALG_CFB_VALUE:
179             // Encrypt the IV into the IV, XOR in the data, and copy to output
180             for(; dSize > 0; dSize -= blockSize)
181             {
182                 // Encrypt the current value of the IV
183                 ENCRYPT(&keySchedule, iv, iv);
184                 pIv = iv;
185                 for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
186                     // XOR the data into the IV to create the cipher text
187                     // and put into the output
188                     *dOut++ = *pIv++ ^= *dIn++;
189             }
190             // If the inner loop (i loop) was smaller than blockSize, then dSize
191             // would have been smaller than blockSize and it is now negative. If
192             // it is negative, then it indicates how many bytes are needed to pad
193             // out the IV for the next round.
194             for(; dSize < 0; dSize++)
195                 *pIv++ = 0;
196             break;
197 #if ALG_ECB
```

```
198             case ALG_ECB_VALUE:
199                 // For ECB the data size must be an even multiple of the
200                 // cipher block size
201                 if((dSize % blockSize) != 0)
202                     return TPM_RC_SIZE;
203                 // Encrypt the input block to the output block
204                 for(; dSize > 0; dSize -= blockSize)
205                 {
206                     ENCRYPT(&keySchedule, dIn, dOut);
207                     dIn = &dIn[blockSize];
208                     dOut = &dOut[blockSize];
209                 }
210                 break;
211     #endif
212             default:
213                 return TPM_RC_FAILURE;
214         }
215         return TPM_RC_SUCCESS;
216     }
```

10.2.19.4.1 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

```
217     LIB_EXPORT TPM_RC
218     CryptSymmetricDecrypt(
219         BYTE                *dOut,          // OUT: decrypted data
220         TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
221         UINT16               keySizeInBits, // IN: key size in bits
222         const BYTE          *key,           // IN: key buffer. The size of this buffer
223                                             //     in bytes is (keySizeInBits + 7) / 8
224         TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
225         TPM_ALG_ID           mode,          // IN: Mode to use
226         INT32                dSize,         // IN: data size (may need to be a
227                                             //     multiple of the blockSize)
228         const BYTE          *dIn            // IN: data buffer
229         )
230     {
231         BYTE                *pIv;
232         int                  i;
233         BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
234         BYTE                *pT;
235         tpmCryptKeySchedule_t        keySchedule;
236         INT16                blockSize;
237         BYTE                *iv;
238         TpmCryptSetSymKeyCall_t      encrypt;
239         TpmCryptSetSymKeyCall_t      decrypt;
240         BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
241
242         // These are used but the compiler can't tell because they are initialized
243         // in case statements and it can't tell if they are always initialized
244         // when needed, so... Comment these out if the compiler can tell or doesn't
245         // care that these are initialized before use.
246         encrypt = NULL;
247         decrypt = NULL;
248
249         pAssert(dOut != NULL && key != NULL && dIn != NULL);
```

```
250         if(dSize == 0)
251             return TPM_RC_SUCCESS;
252 
253         TEST(algorithm);
254         blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
255         if(blockSize == 0)
256             return TPM_RC_FAILURE;
257         // If the iv is provided, then it is expected to be block sized. In some cases,
258         // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
259         // with no knowledge of the actual block size. This function will set it.
260         if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
261         {
262             ivInOut->t.size = blockSize;
263             iv = ivInOut->t.buffer;
264         }
265         else
266             iv = defaultIv;
267 
268         pIv = iv;
269         // Use the mode to select the key schedule to create. Encrypt always uses the
270         // encryption schedule. Depending on the mode, decryption might use either
271         // the decryption or encryption schedule.
272         switch(mode)
273         {
274     #if ALG_CBC || ALG_ECB
275             case ALG_CBC_VALUE: // decrypt = decrypt
276             case ALG_ECB_VALUE:
277                 // For ECB and CBC, the data size must be an even multiple of the
278                 // cipher block size
279                 if((dSize % blockSize) != 0)
280                     return TPM_RC_SIZE;
281                 SELECT(DECRYPT);
282                 break;
283     #endif
284             default:
285                 // For the remaining stream ciphers, use encryption to decrypt
286                 SELECT(ENCRYPT);
287                 break;
288         }
289         // Now do the mode-dependent decryption
290         switch(mode)
291         {
292     #if ALG_CBC
293             case ALG_CBC_VALUE:
294                 // Copy the input data to a temp buffer, decrypt the buffer into the
295                 // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
296                 for(; dSize > 0; dSize -= blockSize)
297                 {
298                     pT = tmp;
299                     for(i = blockSize; i > 0; i--)
300                         *pT++ = *dIn++;
301                     DECRYPT(&keySchedule, tmp, dOut);
302                     pIv = iv;
303                     pT = tmp;
304                     for(i = blockSize; i > 0; i--)
305                     {
306                         *dOut++ ^= *pIv;
307                         *pIv++ = *pT++;
308                     }
309                 }
310                 break;
311     #endif
312             case ALG_CFB_VALUE:
313                 for(; dSize > 0; dSize -= blockSize)
314                 {
315                     // Encrypt the IV into the temp buffer
```

```
316                 ENCRYPT(&keySchedule, iv, tmp);
317                 pT = tmp;
318                 pIv = iv;
319                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
320                     // Copy the current cipher text to IV, XOR
321                     // with the temp buffer and put into the output
322                     *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
323             }
324             // If the inner loop (i loop) was smaller than blockSize, then dSize
325             // would have been smaller than blockSize and it is now negative
326             // If it is negative, then it indicates how may fill bytes
327             // are needed to pad out the IV for the next round.
328             for(; dSize < 0; dSize++)
329                 *pIv++ = 0;
330 
331             break;
332 #if ALG_CTR
333         case ALG_CTR_VALUE:
334             for(; dSize > 0; dSize -= blockSize)
335             {
336                 // Encrypt the current value of the IV(counter)
337                 ENCRYPT(&keySchedule, iv, tmp);
338 
339                 //increment the counter (counter is big-endian so start at end)
340                 for(i = blockSize - 1; i >= 0; i--)
341                     if((iv[i] += 1) != 0)
342                         break;
343                 // XOR the encrypted counter value with input and put into output
344                 pT = tmp;
345                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
346                     *dOut++ = *dIn++ ^ *pT++;
347             }
348             break;
349 #endif
350 #if ALG_ECB
351         case ALG_ECB_VALUE:
352             for(; dSize > 0; dSize -= blockSize)
353             {
354                 DECRYPT(&keySchedule, dIn, dOut);
355                 dIn = &dIn[blockSize];
356                 dOut = &dOut[blockSize];
357             }
358             break;
359 #endif
360 #if ALG_OFB
361         case ALG_OFB_VALUE:
362             // This is written so that dIn and dOut may be the same
363             for(; dSize > 0; dSize -= blockSize)
364             {
365                 // Encrypt the current value of the "IV"
366                 ENCRYPT(&keySchedule, iv, iv);
367 
368                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
369                 pIv = iv;
370                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
371                     *dOut++ = (*pIv++ ^ *dIn++);
372             }
373             break;
374 #endif
375         default:
376             return TPM_RC_FAILURE;
377     }
378     return TPM_RC_SUCCESS;
379 }
```

10.2.19.4.2 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

```
380 TPM_RC
381 CryptSymKeyValidate(
382     TPMT_SYM_DEF_OBJECT *symDef,
383     TPM2B_SYM_KEY       *key
384     )
385 {
386     if(key->t.size != BITS_TO_BYTES(symDef->keyBits.sym))
387         return TPM_RCS_KEY_SIZE;
388 #if ALG_TDES
389     if(symDef->algorithm == TPM_ALG_TDES && !CryptDesValidateKey(key))
390         return TPM_RCS_KEY;
391 #endif // ALG_TDES
392     return TPM_RC_SUCCESS;
393 }
```

10.2.20 PrimeData.c

```
#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.

```
const BN_STRUCT(43 * RADIX_BITS) s_CompositeOfSmallPrimes_ =
{44, 44,
{ 0x2ED42696, 0x2BBFA177, 0x4820594F, 0xF73F4841,
0xBFAC313A, 0xCAC3EB81, 0xF6F26BF8, 0x7FAB5061,
0x59746FB7, 0xF71377F6, 0x3B19855B, 0xCBD03132,
0xBB92EF1B, 0x3AC3152C, 0xE87C8273, 0xC0AE0E69,
0x74A9E295, 0x448CCE86, 0x63CA1907, 0x8A0BF944,
0xF8CC3BE0, 0xC26F0AF5, 0xC501C02F, 0x6579441A,
0xD1099CDA, 0x6BC76A00, 0xC81A3228, 0xBFB1AB25,
0x70FA3841, 0x51B3D076, 0xCC2359ED, 0xD9EE0769,
0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC891,
0x96658ED2, 0x1753EFE5, 0x3AE4A5A6, 0x8FD4A97F,
0x8B15E7EB, 0x0243C3E1, 0xE0F0C31D, 0x0000000B }
};
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().

```
const uint32_t      s_LastPrimeInTable = 65537;
const uint32_t      s_PrimeTableSize = 4097;
const uint32_t      s_PrimesInTable = 6542;
const unsigned char s_PrimeTable[] = {
    0x6e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0x4c, 0x4a, 0x86,
    0x0d, 0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4,
    0x44, 0x11, 0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32,
    0x21, 0x99, 0x34, 0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14,
    0x05, 0x42, 0x30, 0x6c, 0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12,
    0x28, 0x89, 0x04, 0x65, 0x98, 0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80,
    0x21, 0x42, 0x12, 0x41, 0xc9, 0x04, 0x21, 0xc0, 0x32, 0x2d, 0x98, 0x00,
    0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68, 0x82, 0xb0, 0x25, 0x08, 0x22,
    0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90, 0x06, 0x40, 0x43, 0x30,
    0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4, 0x0d, 0x41, 0x12,
    0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04, 0x1b, 0x82,
    0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91, 0x14,
    0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x52, 0x00,
    0x08, 0x10, 0xa2, 0x4c, 0x00, 0x82, 0x01, 0x51, 0x10, 0x08, 0x8b, 0xa4,
    0x25, 0x9a, 0x30, 0x44, 0x81, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80,
    0x08, 0x49, 0x84, 0x20, 0x50, 0x32, 0x00, 0x18, 0xa2, 0x40, 0x11, 0x24,
    0x28, 0x01, 0x84, 0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36,
    0x08, 0x00, 0x26, 0x29, 0x83, 0x82, 0x61, 0xc0, 0x80, 0x04, 0x10, 0x10,
    0x6d, 0x00, 0x22, 0x48, 0x58, 0x26, 0x0c, 0xc2, 0x10, 0x48, 0x89, 0x24,
    0x20, 0x58, 0x20, 0x45, 0x88, 0x24, 0x00, 0x19, 0x02, 0x25, 0xc0, 0x10,
    0x68, 0x08, 0x14, 0x01, 0xca, 0x32, 0x28, 0x80, 0x00, 0x04, 0x4b, 0x26,
    0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0, 0x00, 0x01, 0x10, 0x32,
    0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24, 0x04, 0x48, 0x10,
    0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09, 0x88, 0x24,
    0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80, 0x96,
    0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x58, 0x20,
    0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22,
    0x44, 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0x0a, 0x00,
    0x24, 0x89, 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24,
```

```
50      0x45, 0x08, 0x00, 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04,
51      0x00, 0x8a, 0x90, 0x44, 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0x0a, 0x02,
52      0x01, 0x40, 0x02, 0x28, 0x40, 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86,
53      0x08, 0x42, 0x00, 0x2c, 0x52, 0x04, 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12,
54      0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00, 0x03, 0x14, 0x21, 0x00, 0x10,
55      0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x40, 0x06, 0x09, 0xc3, 0x84,
56      0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90, 0x0c, 0x02, 0xa2,
57      0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41, 0x08, 0x80,
58      0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00, 0x02,
59      0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
60      0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14,
61      0x29, 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x20,
62      0x2d, 0x40, 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24,
63      0x04, 0x59, 0x84, 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02,
64      0x48, 0x81, 0x16, 0x48, 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0,
65      0x40, 0x02, 0x00, 0x04, 0x91, 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06,
66      0x09, 0x48, 0x14, 0x25, 0x92, 0x20, 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86,
67      0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45, 0x08, 0x32, 0x00, 0x98, 0x06,
68      0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81, 0x14, 0x44, 0x82, 0x12,
69      0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04, 0x20, 0x81, 0x24,
70      0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00, 0x00, 0x02,
71      0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18, 0xa4,
72      0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
73      0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12,
74      0x05, 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04,
75      0x01, 0x88, 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80,
76      0x08, 0x40, 0x94, 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02,
77      0x01, 0x42, 0x84, 0x00, 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30,
78      0x44, 0x40, 0x22, 0x21, 0x00, 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20,
79      0x40, 0x18, 0x16, 0x40, 0x40, 0x00, 0x28, 0x52, 0x90, 0x08, 0x82, 0x14,
80      0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40, 0x0a, 0xa0, 0x20, 0x93, 0x80,
81      0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01, 0x10, 0x40, 0x11, 0x06,
82      0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00, 0x28, 0x80, 0x90,
83      0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01, 0x18, 0x00,
84      0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48, 0x14,
85      0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
86      0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82,
87      0x00, 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04,
88      0x00, 0x50, 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12,
89      0x08, 0x00, 0xa4, 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10,
90      0x21, 0x02, 0x20, 0x41, 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00,
91      0x21, 0x49, 0x84, 0x20, 0x10, 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22,
92      0x01, 0x01, 0x84, 0x69, 0xc1, 0x30, 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00,
93      0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61, 0x00, 0xa0, 0x00, 0xc0, 0x30,
94      0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80, 0x94, 0x40, 0x01, 0x84,
95      0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24, 0x00, 0x01, 0x06,
96      0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c, 0x03, 0x80,
97      0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89, 0xa2,
98      0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
99      0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10,
100     0x04, 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84,
101     0x41, 0x89, 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84,
102     0x08, 0xc1, 0x00, 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36,
103     0x04, 0x49, 0xa0, 0x24, 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02,
104     0x01, 0x08, 0x06, 0x08, 0x09, 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80,
105     0x60, 0x00, 0x00, 0x68, 0x01, 0x90, 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80,
106     0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25, 0x81, 0x06, 0x40, 0x49, 0x00,
107     0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18, 0x30, 0x05, 0x01, 0xa6,
108     0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20, 0x00, 0x88, 0x12,
109     0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49, 0x00, 0x04,
110     0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10, 0x00,
111     0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
```

```
112       0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30,
113       0x04, 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20,
114       0x08, 0x10, 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0,
115       0x61, 0x40, 0x10, 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14,
116       0x60, 0x41, 0x80, 0x41, 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20,
117       0x29, 0x52, 0x00, 0x41, 0x08, 0x34, 0x60, 0x10, 0x00, 0x28, 0x01, 0x10,
118       0x40, 0x00, 0x84, 0x08, 0x42, 0x90, 0x20, 0x48, 0x04, 0x04, 0x52, 0x02,
119       0x00, 0x08, 0x20, 0x04, 0x00, 0x82, 0x0d, 0x00, 0x82, 0x40, 0x02, 0x10,
120       0x05, 0x48, 0x20, 0x40, 0x99, 0x00, 0x00, 0x01, 0x06, 0x24, 0xc0, 0x00,
121       0x68, 0x82, 0x04, 0x21, 0x12, 0x10, 0x44, 0x08, 0x04, 0x00, 0x40, 0xa6,
122       0x20, 0xd0, 0x16, 0x09, 0xc9, 0x24, 0x41, 0x02, 0x20, 0x0c, 0x09, 0x92,
123       0x40, 0x12, 0x00, 0x00, 0x40, 0x00, 0x09, 0x43, 0x84, 0x20, 0x98, 0x02,
124       0x01, 0x11, 0x24, 0x00, 0x43, 0x24, 0x00, 0x03, 0x90, 0x08, 0x41, 0x30,
125       0x24, 0x58, 0x20, 0x4c, 0x80, 0x82, 0x08, 0x10, 0x24, 0x25, 0x81, 0x06,
126       0x41, 0x09, 0x10, 0x20, 0x18, 0x10, 0x44, 0x80, 0x10, 0x00, 0x4a, 0x24,
127       0x0d, 0x01, 0x94, 0x28, 0x80, 0x30, 0x00, 0xc0, 0x02, 0x60, 0x10, 0x84,
128       0x0c, 0x02, 0x00, 0x09, 0x02, 0x82, 0x01, 0x08, 0x10, 0x04, 0xc2, 0x20,
129       0x68, 0x09, 0x06, 0x04, 0x18, 0x00, 0x00, 0x11, 0x90, 0x08, 0x0b, 0x10,
130       0x21, 0x82, 0x02, 0x0c, 0x10, 0xb6, 0x08, 0x00, 0x26, 0x00, 0x41, 0x02,
131       0x01, 0x4a, 0x24, 0x21, 0x1a, 0x20, 0x24, 0x80, 0x00, 0x44, 0x02, 0x00,
132       0x2d, 0x40, 0x02, 0x00, 0x8b, 0x94, 0x20, 0x10, 0x00, 0x20, 0x90, 0xa6,
133       0x40, 0x13, 0x00, 0x2c, 0x11, 0x86, 0x61, 0x01, 0x80, 0x41, 0x10, 0x02,
134       0x04, 0x81, 0x30, 0x48, 0x48, 0x20, 0x28, 0x50, 0x80, 0x21, 0x8a, 0x10,
135       0x04, 0x08, 0x10, 0x09, 0x10, 0x10, 0x48, 0x42, 0xa0, 0x0c, 0x82, 0x92,
136       0x60, 0xc0, 0x20, 0x05, 0xd2, 0x20, 0x40, 0x01, 0x00, 0x04, 0x08, 0x82,
137       0x2d, 0x82, 0x02, 0x00, 0x48, 0x80, 0x41, 0x48, 0x10, 0x00, 0x91, 0x04,
138       0x04, 0x03, 0x84, 0x00, 0xc2, 0x04, 0x68, 0x00, 0x00, 0x64, 0xc0, 0x22,
139       0x40, 0x08, 0x32, 0x44, 0x09, 0x86, 0x00, 0x91, 0x02, 0x28, 0x01, 0x00,
140       0x64, 0x48, 0x00, 0x24, 0x10, 0x90, 0x00, 0x43, 0x00, 0x21, 0x52, 0x86,
141       0x41, 0x8b, 0x90, 0x20, 0x40, 0x20, 0x08, 0x88, 0x04, 0x44, 0x13, 0x20,
142       0x00, 0x02, 0x84, 0x60, 0x81, 0x90, 0x24, 0x40, 0x30, 0x00, 0x08, 0x10,
143       0x08, 0x08, 0x02, 0x01, 0x10, 0x04, 0x20, 0x43, 0xb4, 0x40, 0x90, 0x12,
144       0x68, 0x01, 0x80, 0x4c, 0x18, 0x00, 0x08, 0xc0, 0x12, 0x49, 0x40, 0x10,
145       0x24, 0x1a, 0x00, 0x41, 0x89, 0x24, 0x4c, 0x10, 0x00, 0x04, 0x52, 0x10,
146       0x09, 0x4a, 0x20, 0x41, 0x48, 0x22, 0x69, 0x11, 0x14, 0x08, 0x10, 0x06,
147       0x24, 0x80, 0x84, 0x28, 0x00, 0x10, 0x00, 0x40, 0x10, 0x01, 0x08, 0x26,
148       0x08, 0x48, 0x06, 0x28, 0x00, 0x14, 0x01, 0x42, 0x84, 0x04, 0x0a, 0x20,
149       0x00, 0x01, 0x82, 0x08, 0x00, 0x82, 0x24, 0x12, 0x04, 0x40, 0x40, 0xa0,
150       0x40, 0x90, 0x10, 0x04, 0x90, 0x22, 0x40, 0x10, 0x20, 0x2c, 0x80, 0x10,
151       0x28, 0x43, 0x00, 0x04, 0x58, 0x00, 0x01, 0x81, 0x10, 0x48, 0x09, 0x20,
152       0x21, 0x83, 0x04, 0x00, 0x42, 0xa4, 0x44, 0x00, 0x00, 0x6c, 0x10, 0xa0,
153       0x44, 0x48, 0x80, 0x00, 0x83, 0x80, 0x48, 0xc9, 0x00, 0x00, 0x00, 0x02,
154       0x05, 0x10, 0xb0, 0x04, 0x13, 0x04, 0x29, 0x10, 0x92, 0x40, 0x08, 0x04,
155       0x44, 0x82, 0x22, 0x00, 0x19, 0x20, 0x00, 0x19, 0x20, 0x01, 0x81, 0x90,
156       0x60, 0x8a, 0x00, 0x41, 0xc0, 0x02, 0x45, 0x10, 0x04, 0x00, 0x02, 0xa2,
157       0x09, 0x40, 0x10, 0x21, 0x49, 0x20, 0x01, 0x42, 0x30, 0x2c, 0x00, 0x14,
158       0x44, 0x01, 0x22, 0x04, 0x02, 0x92, 0x08, 0x89, 0x04, 0x21, 0x80, 0x10,
159       0x05, 0x01, 0x20, 0x40, 0x41, 0x80, 0x04, 0x00, 0x12, 0x09, 0x40, 0xb0,
160       0x64, 0x58, 0x32, 0x01, 0x08, 0x90, 0x00, 0x41, 0x04, 0x09, 0xc1, 0x80,
161       0x61, 0x08, 0x90, 0x00, 0x9a, 0x00, 0x24, 0x01, 0x12, 0x08, 0x02, 0x26,
162       0x05, 0x82, 0x06, 0x08, 0x08, 0x00, 0x20, 0x48, 0x20, 0x00, 0x18, 0x24,
163       0x48, 0x03, 0x02, 0x00, 0x11, 0x00, 0x09, 0x00, 0x84, 0x01, 0x4a, 0x10,
164       0x01, 0x98, 0x00, 0x04, 0x18, 0x86, 0x00, 0xc0, 0x00, 0x20, 0x81, 0x80,
165       0x04, 0x10, 0x30, 0x05, 0x00, 0xb4, 0x0c, 0x4a, 0x82, 0x29, 0x91, 0x02,
166       0x28, 0x00, 0x20, 0x44, 0xc0, 0x00, 0x2c, 0x91, 0x80, 0x40, 0x01, 0xa2,
167       0x00, 0x12, 0x04, 0x09, 0xc3, 0x20, 0x00, 0x08, 0x02, 0x0c, 0x10, 0x22,
168       0x04, 0x00, 0x00, 0x2c, 0x11, 0x86, 0x00, 0xc0, 0x00, 0x00, 0x12, 0x32,
169       0x40, 0x89, 0x80, 0x40, 0x40, 0x02, 0x05, 0x50, 0x86, 0x60, 0x82, 0xa4,
170       0x60, 0x0a, 0x12, 0x4d, 0x80, 0x90, 0x08, 0x12, 0x80, 0x09, 0x02, 0x14,
171       0x48, 0x01, 0x24, 0x20, 0x8a, 0x00, 0x44, 0x90, 0x04, 0x04, 0x01, 0x02,
172       0x00, 0xd1, 0x12, 0x00, 0x0a, 0x04, 0x40, 0x00, 0x32, 0x21, 0x81, 0x24,
173       0x08, 0x19, 0x84, 0x20, 0x02, 0x04, 0x08, 0x89, 0x80, 0x24, 0x02, 0x02,
```

```
174       0x68, 0x18, 0x82, 0x44, 0x42, 0x00, 0x21, 0x40, 0x00, 0x28, 0x01, 0x80,
175       0x45, 0x82, 0x20, 0x40, 0x11, 0x80, 0x0c, 0x02, 0x00, 0x24, 0x40, 0x90,
176       0x01, 0x40, 0x20, 0x20, 0x50, 0x20, 0x28, 0x19, 0x00, 0x40, 0x09, 0x20,
177       0x08, 0x80, 0x04, 0x60, 0x40, 0x80, 0x20, 0x08, 0x30, 0x49, 0x09, 0x34,
178       0x00, 0x11, 0x24, 0x24, 0x82, 0x00, 0x41, 0xc2, 0x00, 0x04, 0x92, 0x02,
179       0x24, 0x80, 0x00, 0x0c, 0x02, 0xa0, 0x00, 0x01, 0x06, 0x60, 0x41, 0x04,
180       0x21, 0xd0, 0x00, 0x01, 0x01, 0x00, 0x48, 0x12, 0x84, 0x04, 0x91, 0x12,
181       0x08, 0x00, 0x24, 0x44, 0x00, 0x12, 0x41, 0x18, 0x26, 0x0c, 0x41, 0x80,
182       0x00, 0x52, 0x04, 0x20, 0x09, 0x00, 0x24, 0x90, 0x20, 0x48, 0x18, 0x02,
183       0x00, 0x03, 0xa2, 0x09, 0xd0, 0x14, 0x00, 0x8a, 0x84, 0x25, 0x4a, 0x00,
184       0x20, 0x98, 0x14, 0x40, 0x00, 0xa2, 0x05, 0x00, 0x00, 0x00, 0x40, 0x14,
185       0x01, 0x58, 0x20, 0x2c, 0x80, 0x84, 0x00, 0x09, 0x20, 0x20, 0x91, 0x02,
186       0x08, 0x02, 0xb0, 0x41, 0x08, 0x30, 0x00, 0x09, 0x10, 0x00, 0x18, 0x02,
187       0x21, 0x02, 0x02, 0x00, 0x00, 0x24, 0x44, 0x08, 0x12, 0x60, 0x00, 0xb2,
188       0x44, 0x12, 0x02, 0x0c, 0xc0, 0x80, 0x40, 0xc8, 0x20, 0x04, 0x50, 0x20,
189       0x05, 0x00, 0xb0, 0x04, 0x0b, 0x04, 0x29, 0x53, 0x00, 0x61, 0x48, 0x30,
190       0x00, 0x82, 0x20, 0x29, 0x00, 0x16, 0x00, 0x53, 0x22, 0x20, 0x43, 0x10,
191       0x48, 0x00, 0x80, 0x04, 0xd2, 0x00, 0x40, 0x00, 0xa2, 0x44, 0x03, 0x80,
192       0x29, 0x00, 0x04, 0x08, 0xc0, 0x04, 0x64, 0x40, 0x30, 0x28, 0x09, 0x84,
193       0x44, 0x50, 0x80, 0x21, 0x02, 0x92, 0x00, 0xc0, 0x10, 0x60, 0x88, 0x22,
194       0x08, 0x80, 0x00, 0x00, 0x18, 0x84, 0x04, 0x83, 0x96, 0x00, 0x81, 0x20,
195       0x05, 0x02, 0x00, 0x45, 0x88, 0x84, 0x00, 0x51, 0x20, 0x20, 0x51, 0x86,
196       0x41, 0x4b, 0x94, 0x00, 0x80, 0x00, 0x08, 0x11, 0x20, 0x4c, 0x58, 0x80,
197       0x04, 0x03, 0x06, 0x20, 0x89, 0x00, 0x05, 0x08, 0x22, 0x05, 0x90, 0x00,
198       0x40, 0x00, 0x82, 0x09, 0x50, 0x00, 0x00, 0x00, 0xa0, 0x41, 0xc2, 0x20,
199       0x08, 0x00, 0x16, 0x08, 0x40, 0x26, 0x21, 0xd0, 0x90, 0x08, 0x81, 0x90,
200       0x41, 0x00, 0x02, 0x44, 0x08, 0x10, 0x0c, 0x0a, 0x86, 0x09, 0x90, 0x04,
201       0x00, 0xc8, 0xa0, 0x04, 0x08, 0x30, 0x20, 0x89, 0x84, 0x00, 0x11, 0x22,
202       0x2c, 0x40, 0x00, 0x08, 0x02, 0xb0, 0x01, 0x48, 0x02, 0x01, 0x09, 0x20,
203       0x04, 0x03, 0x04, 0x00, 0x80, 0x02, 0x60, 0x42, 0x30, 0x21, 0x4a, 0x10,
204       0x44, 0x09, 0x02, 0x00, 0x01, 0x24, 0x00, 0x12, 0x82, 0x21, 0x80, 0xa4,
205       0x20, 0x10, 0x02, 0x04, 0x91, 0xa0, 0x40, 0x18, 0x04, 0x00, 0x02, 0x06,
206       0x69, 0x09, 0x00, 0x05, 0x58, 0x02, 0x01, 0x00, 0x00, 0x48, 0x00, 0x00,
207       0x00, 0x03, 0x92, 0x20, 0x00, 0x34, 0x01, 0xc8, 0x20, 0x48, 0x08, 0x30,
208       0x08, 0x42, 0x80, 0x20, 0x91, 0x90, 0x68, 0x01, 0x04, 0x40, 0x12, 0x02,
209       0x61, 0x00, 0x12, 0x08, 0x01, 0xa0, 0x00, 0x11, 0x04, 0x21, 0x48, 0x04,
210       0x24, 0x92, 0x00, 0x0c, 0x01, 0x84, 0x04, 0x00, 0x00, 0x01, 0x12, 0x96,
211       0x40, 0x01, 0xa0, 0x41, 0x88, 0x22, 0x28, 0x88, 0x00, 0x44, 0x42, 0x80,
212       0x24, 0x12, 0x14, 0x01, 0x42, 0x90, 0x60, 0x1a, 0x10, 0x04, 0x81, 0x10,
213       0x48, 0x08, 0x06, 0x29, 0x83, 0x02, 0x40, 0x02, 0x24, 0x64, 0x80, 0x10,
214       0x05, 0x80, 0x10, 0x40, 0x02, 0x02, 0x08, 0x42, 0x84, 0x01, 0x09, 0x20,
215       0x04, 0x50, 0x00, 0x60, 0x11, 0x30, 0x40, 0x13, 0x02, 0x04, 0x81, 0x00,
216       0x09, 0x08, 0x20, 0x45, 0x4a, 0x10, 0x61, 0x90, 0x26, 0x0c, 0x08, 0x02,
217       0x21, 0x91, 0x00, 0x60, 0x02, 0x04, 0x00, 0x02, 0x00, 0x0c, 0x08, 0x06,
218       0x08, 0x48, 0x84, 0x08, 0x11, 0x02, 0x00, 0x80, 0xa4, 0x00, 0x5a, 0x20,
219       0x00, 0x88, 0x04, 0x04, 0x02, 0x00, 0x09, 0x00, 0x14, 0x08, 0x49, 0x14,
220       0x20, 0xc8, 0x00, 0x04, 0x91, 0xa0, 0x40, 0x59, 0x80, 0x00, 0x12, 0x10,
221       0x00, 0x80, 0x80, 0x65, 0x00, 0x00, 0x04, 0x00, 0x80, 0x40, 0x19, 0x00,
222       0x21, 0x03, 0x84, 0x60, 0xc0, 0x04, 0x24, 0x1a, 0x12, 0x61, 0x80, 0x80,
223       0x08, 0x02, 0x04, 0x09, 0x42, 0x12, 0x20, 0x08, 0x34, 0x04, 0x90, 0x20,
224       0x01, 0x01, 0xa0, 0x00, 0x0b, 0x00, 0x08, 0x91, 0x92, 0x40, 0x02, 0x34,
225       0x40, 0x88, 0x10, 0x61, 0x19, 0x02, 0x00, 0x40, 0x04, 0x25, 0xc0, 0x80,
226       0x68, 0x08, 0x04, 0x21, 0x80, 0x22, 0x04, 0x00, 0xa0, 0x0c, 0x01, 0x84,
227       0x20, 0x41, 0x00, 0x08, 0x8a, 0x00, 0x20, 0x8a, 0x00, 0x48, 0x88, 0x04,
228       0x04, 0x11, 0x82, 0x08, 0x40, 0x86, 0x09, 0x49, 0xa4, 0x40, 0x00, 0x10,
229       0x01, 0x01, 0xa2, 0x04, 0x50, 0x80, 0x0c, 0x80, 0x00, 0x48, 0x82, 0xa0,
230       0x01, 0x18, 0x12, 0x41, 0x01, 0x04, 0x48, 0x41, 0x00, 0x24, 0x01, 0x00,
231       0x00, 0x88, 0x14, 0x00, 0x02, 0x00, 0x68, 0x01, 0x20, 0x08, 0x4a, 0x22,
232       0x08, 0x83, 0x80, 0x00, 0x89, 0x04, 0x01, 0xc2, 0x00, 0x00, 0x00, 0x34,
233       0x04, 0x00, 0x82, 0x28, 0x02, 0x02, 0x41, 0x4a, 0x90, 0x05, 0x82, 0x02,
234       0x09, 0x80, 0x24, 0x04, 0x41, 0x00, 0x01, 0x92, 0x80, 0x28, 0x01, 0x14,
235       0x00, 0x50, 0x20, 0x4c, 0x10, 0xb0, 0x04, 0x43, 0xa4, 0x21, 0x90, 0x04,
```

```
236       0x01, 0x02, 0x00, 0x44, 0x48, 0x00, 0x64, 0x08, 0x06, 0x00, 0x42, 0x20,
237       0x08, 0x02, 0x92, 0x01, 0x4a, 0x00, 0x20, 0x50, 0x32, 0x25, 0x90, 0x22,
238       0x04, 0x09, 0x00, 0x08, 0x11, 0x80, 0x21, 0x01, 0x10, 0x05, 0x00, 0x32,
239       0x08, 0x88, 0x94, 0x08, 0x08, 0x24, 0x0d, 0xc1, 0x80, 0x40, 0x0b, 0x20,
240       0x40, 0x18, 0x12, 0x04, 0x00, 0x22, 0x40, 0x10, 0x26, 0x05, 0xc1, 0x82,
241       0x00, 0x01, 0x30, 0x24, 0x02, 0x22, 0x41, 0x08, 0x24, 0x48, 0x1a, 0x00,
242       0x25, 0xd2, 0x12, 0x28, 0x42, 0x00, 0x04, 0x40, 0x30, 0x41, 0x00, 0x02,
243       0x00, 0x13, 0x20, 0x24, 0xd1, 0x84, 0x08, 0x89, 0x80, 0x04, 0x52, 0x00,
244       0x44, 0x18, 0xa4, 0x00, 0x00, 0x06, 0x20, 0x91, 0x10, 0x09, 0x42, 0x20,
245       0x24, 0x40, 0x30, 0x28, 0x00, 0x84, 0x40, 0x40, 0x80, 0x08, 0x10, 0x04,
246       0x09, 0x08, 0x04, 0x40, 0x08, 0x22, 0x00, 0x19, 0x02, 0x00, 0x00, 0x80,
247       0x2c, 0x02, 0x02, 0x21, 0x01, 0x90, 0x20, 0x40, 0x00, 0x0c, 0x00, 0x34,
248       0x48, 0x58, 0x20, 0x01, 0x43, 0x04, 0x20, 0x80, 0x14, 0x00, 0x90, 0x00,
249       0x6d, 0x11, 0x00, 0x00, 0x40, 0x20, 0x00, 0x03, 0x10, 0x40, 0x88, 0x30,
250       0x05, 0x4a, 0x00, 0x65, 0x10, 0x24, 0x08, 0x18, 0x84, 0x28, 0x03, 0x80,
251       0x20, 0x42, 0xb0, 0x40, 0x00, 0x10, 0x69, 0x19, 0x04, 0x00, 0x00, 0x80,
252       0x04, 0xc2, 0x04, 0x00, 0x01, 0x00, 0x05, 0x00, 0x22, 0x25, 0x08, 0x96,
253       0x04, 0x02, 0x22, 0x00, 0xd0, 0x10, 0x29, 0x01, 0xa0, 0x60, 0x08, 0x10,
254       0x04, 0x01, 0x16, 0x44, 0x10, 0x02, 0x28, 0x02, 0x82, 0x48, 0x40, 0x84,
255       0x20, 0x90, 0x22, 0x28, 0x80, 0x04, 0x00, 0x40, 0x04, 0x24, 0x00, 0x80,
256       0x29, 0x03, 0x10, 0x60, 0x48, 0x00, 0x00, 0x81, 0xa0, 0x00, 0x51, 0x20,
257       0x0c, 0xd1, 0x00, 0x01, 0x41, 0x20, 0x04, 0x92, 0x00, 0x00, 0x10, 0x92,
258       0x00, 0x42, 0x04, 0x05, 0x01, 0x86, 0x40, 0x80, 0x10, 0x20, 0x52, 0x20,
259       0x21, 0x00, 0x10, 0x48, 0x0a, 0x02, 0x00, 0xd0, 0x12, 0x41, 0x48, 0x80,
260       0x04, 0x00, 0x00, 0x48, 0x09, 0x22, 0x04, 0x00, 0x24, 0x00, 0x43, 0x10,
261       0x60, 0x0a, 0x00, 0x44, 0x12, 0x20, 0x2c, 0x08, 0x20, 0x44, 0x00, 0x84,
262       0x09, 0x40, 0x06, 0x08, 0xc1, 0x00, 0x40, 0x80, 0x20, 0x00, 0x98, 0x12,
263       0x48, 0x10, 0xa2, 0x20, 0x00, 0x84, 0x48, 0xc0, 0x10, 0x20, 0x90, 0x12,
264       0x08, 0x98, 0x82, 0x00, 0x0a, 0xa0, 0x04, 0x03, 0x00, 0x28, 0xc3, 0x00,
265       0x44, 0x42, 0x10, 0x04, 0x08, 0x04, 0x40, 0x00, 0x00, 0x05, 0x10, 0x00,
266       0x21, 0x03, 0x80, 0x04, 0x88, 0x12, 0x69, 0x10, 0x00, 0x04, 0x08, 0x04,
267       0x04, 0x02, 0x84, 0x48, 0x49, 0x04, 0x20, 0x18, 0x02, 0x64, 0x80, 0x30,
268       0x08, 0x01, 0x02, 0x00, 0x52, 0x12, 0x49, 0x08, 0x20, 0x41, 0x88, 0x10,
269       0x48, 0x08, 0x34, 0x00, 0x01, 0x86, 0x05, 0xd0, 0x00, 0x00, 0x83, 0x84,
270       0x21, 0x40, 0x02, 0x41, 0x10, 0x80, 0x48, 0x40, 0xa2, 0x20, 0x51, 0x00,
271       0x00, 0x49, 0x00, 0x01, 0x90, 0x20, 0x40, 0x18, 0x02, 0x40, 0x02, 0x22,
272       0x05, 0x40, 0x80, 0x08, 0x82, 0x10, 0x20, 0x18, 0x00, 0x05, 0x01, 0x82,
273       0x40, 0x58, 0x00, 0x04, 0x81, 0x90, 0x29, 0x01, 0xa0, 0x64, 0x00, 0x22,
274       0x40, 0x01, 0xa2, 0x00, 0x18, 0x04, 0x0d, 0x00, 0x00, 0x60, 0x80, 0x94,
275       0x60, 0x82, 0x10, 0x0d, 0x80, 0x30, 0x0c, 0x12, 0x20, 0x00, 0x00, 0x12,
276       0x40, 0xc0, 0x20, 0x21, 0x58, 0x02, 0x41, 0x10, 0x80, 0x44, 0x03, 0x02,
277       0x04, 0x13, 0x90, 0x29, 0x08, 0x00, 0x44, 0xc0, 0x00, 0x21, 0x00, 0x26,
278       0x00, 0x1a, 0x80, 0x01, 0x13, 0x14, 0x20, 0x0a, 0x14, 0x20, 0x00, 0x32,
279       0x61, 0x08, 0x00, 0x40, 0x42, 0x20, 0x09, 0x80, 0x06, 0x01, 0x81, 0x80,
280       0x60, 0x42, 0x00, 0x68, 0x90, 0x82, 0x08, 0x42, 0x80, 0x04, 0x02, 0x80,
281       0x09, 0x0b, 0x04, 0x00, 0x98, 0x00, 0x0c, 0x81, 0x06, 0x44, 0x48, 0x84,
282       0x28, 0x03, 0x92, 0x00, 0x01, 0x80, 0x40, 0x0a, 0x00, 0x0c, 0x81, 0x02,
283       0x08, 0x51, 0x04, 0x28, 0x90, 0x02, 0x20, 0x09, 0x10, 0x60, 0x00, 0x00,
284       0x09, 0x81, 0xa0, 0x0c, 0x00, 0xa4, 0x09, 0x00, 0x02, 0x28, 0x80, 0x20,
285       0x00, 0x02, 0x02, 0x04, 0x81, 0x14, 0x04, 0x00, 0x04, 0x09, 0x11, 0x12,
286       0x60, 0x40, 0x20, 0x01, 0x48, 0x30, 0x40, 0x11, 0x00, 0x08, 0x0a, 0x86,
287       0x00, 0x00, 0x04, 0x60, 0x81, 0x04, 0x01, 0xd0, 0x02, 0x41, 0x18, 0x90,
288       0x00, 0x0a, 0x20, 0x00, 0xc1, 0x06, 0x01, 0x08, 0x80, 0x64, 0xca, 0x10,
289       0x04, 0x99, 0x80, 0x48, 0x01, 0x82, 0x20, 0x50, 0x90, 0x48, 0x80, 0x84,
290       0x20, 0x90, 0x22, 0x00, 0x19, 0x00, 0x04, 0x18, 0x20, 0x24, 0x10, 0x86,
291       0x40, 0xc2, 0x00, 0x24, 0x12, 0x10, 0x44, 0x00, 0x16, 0x08, 0x10, 0x24,
292       0x00, 0x12, 0x06, 0x01, 0x08, 0x90, 0x00, 0x12, 0x02, 0x4d, 0x10, 0x80,
293       0x40, 0x50, 0x22, 0x00, 0x43, 0x10, 0x01, 0x00, 0x30, 0x21, 0x0a, 0x00,
294       0x00, 0x01, 0x14, 0x00, 0x10, 0x84, 0x04, 0xc1, 0x10, 0x29, 0x0a, 0x00,
295       0x01, 0x8a, 0x00, 0x20, 0x01, 0x12, 0x0c, 0x49, 0x20, 0x04, 0x81, 0x00,
296       0x48, 0x01, 0x04, 0x60, 0x80, 0x12, 0x0c, 0x08, 0x10, 0x48, 0x4a, 0x04,
297       0x28, 0x10, 0x00, 0x28, 0x40, 0x84, 0x45, 0x50, 0x10, 0x60, 0x10, 0x06,
```

```
298       0x44, 0x01, 0x80, 0x09, 0x00, 0x86, 0x01, 0x42, 0xa0, 0x00, 0x90, 0x00,
299       0x05, 0x90, 0x22, 0x40, 0x41, 0x00, 0x08, 0x80, 0x02, 0x08, 0xc0, 0x00,
300       0x01, 0x58, 0x30, 0x49, 0x09, 0x14, 0x00, 0x41, 0x02, 0x0c, 0x02, 0x80,
301       0x40, 0x89, 0x00, 0x24, 0x08, 0x10, 0x05, 0x90, 0x32, 0x40, 0x0a, 0x82,
302       0x08, 0x00, 0x12, 0x61, 0x00, 0x04, 0x21, 0x00, 0x22, 0x04, 0x10, 0x24,
303       0x08, 0x0a, 0x04, 0x01, 0x10, 0x00, 0x20, 0x40, 0x84, 0x04, 0x88, 0x22,
304       0x20, 0x90, 0x12, 0x00, 0x53, 0x06, 0x24, 0x01, 0x04, 0x40, 0x0b, 0x14,
305       0x60, 0x82, 0x02, 0x0d, 0x10, 0x90, 0x0c, 0x08, 0x20, 0x09, 0x00, 0x14,
306       0x09, 0x80, 0x80, 0x24, 0x82, 0x00, 0x40, 0x01, 0x02, 0x44, 0x01, 0x20,
307       0x0c, 0x40, 0x84, 0x40, 0x0a, 0x10, 0x41, 0x00, 0x30, 0x05, 0x09, 0x80,
308       0x44, 0x08, 0x20, 0x20, 0x02, 0x00, 0x49, 0x43, 0x20, 0x21, 0x00, 0x20,
309       0x00, 0x01, 0xb6, 0x08, 0x40, 0x04, 0x08, 0x02, 0x80, 0x01, 0x41, 0x80,
310       0x40, 0x08, 0x10, 0x24, 0x00, 0x20, 0x04, 0x12, 0x86, 0x09, 0xc0, 0x12,
311       0x21, 0x81, 0x14, 0x04, 0x00, 0x02, 0x20, 0x89, 0xb4, 0x44, 0x12, 0x80,
312       0x00, 0xd1, 0x00, 0x69, 0x40, 0x80, 0x00, 0x42, 0x12, 0x00, 0x18, 0x04,
313       0x00, 0x49, 0x06, 0x21, 0x02, 0x04, 0x28, 0x02, 0x84, 0x01, 0xc0, 0x10,
314       0x68, 0x00, 0x20, 0x08, 0x40, 0x00, 0x08, 0x91, 0x10, 0x01, 0x81, 0x24,
315       0x04, 0xd2, 0x10, 0x4c, 0x88, 0x86, 0x00, 0x10, 0x80, 0x0c, 0x02, 0x14,
316       0x00, 0x8a, 0x90, 0x40, 0x18, 0x20, 0x21, 0x80, 0xa4, 0x00, 0x58, 0x24,
317       0x20, 0x10, 0x10, 0x60, 0xc1, 0x30, 0x41, 0x48, 0x02, 0x48, 0x09, 0x00,
318       0x40, 0x09, 0x02, 0x05, 0x11, 0x82, 0x20, 0x4a, 0x20, 0x24, 0x18, 0x02,
319       0x0c, 0x10, 0x22, 0x0c, 0x0a, 0x04, 0x00, 0x03, 0x06, 0x48, 0x48, 0x04,
320       0x04, 0x02, 0x00, 0x21, 0x80, 0x84, 0x00, 0x18, 0x00, 0x0c, 0x02, 0x12,
321       0x01, 0x00, 0x14, 0x05, 0x82, 0x10, 0x41, 0x89, 0x12, 0x08, 0x40, 0xa4,
322       0x21, 0x01, 0x84, 0x48, 0x02, 0x10, 0x60, 0x40, 0x02, 0x28, 0x00, 0x14,
323       0x08, 0x40, 0xa0, 0x20, 0x51, 0x12, 0x00, 0xc2, 0x00, 0x01, 0x1a, 0x30,
324       0x40, 0x89, 0x12, 0x4c, 0x02, 0x80, 0x00, 0x00, 0x14, 0x01, 0x01, 0xa0,
325       0x21, 0x18, 0x22, 0x21, 0x18, 0x06, 0x40, 0x01, 0x80, 0x00, 0x90, 0x04,
326       0x48, 0x02, 0x30, 0x04, 0x08, 0x00, 0x05, 0x88, 0x24, 0x08, 0x48, 0x04,
327       0x24, 0x02, 0x06, 0x00, 0x80, 0x00, 0x00, 0x00, 0x10, 0x65, 0x11, 0x90,
328       0x00, 0x0a, 0x82, 0x04, 0xc3, 0x04, 0x60, 0x48, 0x24, 0x04, 0x92, 0x02,
329       0x44, 0x88, 0x80, 0x40, 0x18, 0x06, 0x29, 0x80, 0x10, 0x01, 0x00, 0x00,
330       0x44, 0xc8, 0x10, 0x21, 0x89, 0x30, 0x00, 0x4b, 0xa0, 0x01, 0x10, 0x14,
331       0x00, 0x02, 0x94, 0x40, 0x00, 0x20, 0x65, 0x00, 0xa2, 0x0c, 0x40, 0x22,
332       0x20, 0x81, 0x12, 0x20, 0x82, 0x04, 0x01, 0x10, 0x00, 0x08, 0x88, 0x00,
333       0x00, 0x11, 0x80, 0x04, 0x42, 0x80, 0x40, 0x41, 0x14, 0x00, 0x40, 0x32,
334       0x2c, 0x80, 0x24, 0x04, 0x19, 0x00, 0x00, 0x91, 0x00, 0x20, 0x83, 0x00,
335       0x05, 0x40, 0x20, 0x09, 0x01, 0x84, 0x40, 0x40, 0x20, 0x20, 0x11, 0x00,
336       0x40, 0x41, 0x90, 0x20, 0x00, 0x00, 0x40, 0x90, 0x92, 0x48, 0x18, 0x06,
337       0x08, 0x81, 0x80, 0x48, 0x01, 0x34, 0x24, 0x10, 0x20, 0x04, 0x00, 0x20,
338       0x04, 0x18, 0x06, 0x2d, 0x90, 0x10, 0x01, 0x00, 0x90, 0x00, 0x0a, 0x22,
339       0x01, 0x00, 0x22, 0x00, 0x11, 0x84, 0x01, 0x01, 0x00, 0x20, 0x88, 0x00,
340       0x44, 0x00, 0x22, 0x01, 0x00, 0xa6, 0x40, 0x02, 0x06, 0x20, 0x11, 0x00,
341       0x01, 0xc8, 0xa0, 0x04, 0x8a, 0x00, 0x28, 0x19, 0x80, 0x00, 0x52, 0xa0,
342       0x24, 0x12, 0x12, 0x09, 0x08, 0x24, 0x01, 0x48, 0x00, 0x04, 0x00, 0x24,
343       0x40, 0x02, 0x84, 0x08, 0x00, 0x04, 0x48, 0x40, 0x90, 0x60, 0x0a, 0x22,
344       0x01, 0x88, 0x14, 0x08, 0x01, 0x02, 0x08, 0xd3, 0x00, 0x20, 0xc0, 0x90,
345       0x24, 0x10, 0x00, 0x00, 0x01, 0xb0, 0x08, 0x0a, 0xa0, 0x00, 0x80, 0x00,
346       0x01, 0x09, 0x00, 0x20, 0x52, 0x02, 0x25, 0x00, 0x24, 0x04, 0x02, 0x84,
347       0x24, 0x10, 0x92, 0x40, 0x02, 0xa0, 0x40, 0x00, 0x22, 0x08, 0x11, 0x04,
348       0x08, 0x01, 0x22, 0x00, 0x42, 0x14, 0x00, 0x09, 0x90, 0x21, 0x00, 0x30,
349       0x6c, 0x00, 0x00, 0x0c, 0x00, 0x22, 0x09, 0x90, 0x10, 0x28, 0x40, 0x00,
350       0x20, 0xc0, 0x20, 0x00, 0x90, 0x00, 0x40, 0x01, 0x82, 0x05, 0x12, 0x12,
351       0x09, 0xc1, 0x04, 0x61, 0x80, 0x02, 0x28, 0x81, 0x24, 0x00, 0x49, 0x04,
352       0x08, 0x10, 0x86, 0x29, 0x41, 0x80, 0x21, 0x0a, 0x30, 0x49, 0x88, 0x90,
353       0x00, 0x41, 0x04, 0x29, 0x81, 0x80, 0x41, 0x09, 0x00, 0x40, 0x12, 0x10,
354       0x40, 0x00, 0x10, 0x40, 0x48, 0x02, 0x05, 0x80, 0x02, 0x21, 0x40, 0x20,
355       0x00, 0x58, 0x20, 0x60, 0x00, 0x90, 0x48, 0x00, 0x80, 0x28, 0xc0, 0x80,
356       0x48, 0x00, 0x00, 0x44, 0x80, 0x02, 0x00, 0x09, 0x06, 0x00, 0x12, 0x02,
357       0x01, 0x00, 0x10, 0x08, 0x83, 0x10, 0x45, 0x12, 0x00, 0x2c, 0x08, 0x04,
358       0x44, 0x00, 0x20, 0x20, 0xc0, 0x10, 0x20, 0x01, 0x00, 0x05, 0xc8, 0x20,
359       0x04, 0x98, 0x10, 0x08, 0x10, 0x00, 0x24, 0x02, 0x16, 0x40, 0x88, 0x00,
```

```
360         0x61, 0x88, 0x12, 0x24, 0x80, 0xa6, 0x00, 0x42, 0x00, 0x08, 0x10, 0x06,
361         0x48, 0x40, 0xa0, 0x00, 0x50, 0x20, 0x04, 0x81, 0xa4, 0x40, 0x18, 0x00,
362         0x08, 0x10, 0x80, 0x01, 0x01};
363
364     #if RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT
365     UINT32  PrimeIndex = 0;
366     UINT32  failedAtIteration[10] = {0};
367     UINT32  PrimeCounts[3] = {0};
368     UINT32  MillerRabinTrials[3] = {0};
369     UINT32  totalFieldsSieved[3] = {0};
370     UINT32  bitsInFieldAfterSieve[3] = {0};
371     UINT32  emptyFieldsSieved[3] = {0};
372     UINT32  noPrimeFields[3] = {0};
373     UINT32  primesChecked[3] = {0};
374     UINT16  lastSievePrime = 0;
375     #endif
```

10.2.21 RsaKeyCache.c

10.2.21.1 Introduction

This file contains the functions to implement the RSA key cache that can be used to speed up simulation.

Only one key is created for each supported key size and it is returned whenever a key of that size is requested.

If desired, the key cache can be populated from a file. This allows multiple TPM to run with the same RSA keys. Also, when doing simulation, the DRBG will use preset sequences so it is not too hard to repeat sequences for debug or profile or stress.

When the key cache is enabled, a call to CryptRsaGenerateKey() will call the GetCachedRsaKey(). If the cache is enabled and populated, then the cached key of the requested size is returned. If a key of the requested size is not available, the no key is loaded and the requested key will need to be generated. If the cache is not populated, the TPM will open a file that has the appropriate name for the type of keys required (CRT or no-CRT). If the file is the right size, it is used. If the file doesn't exist or the file does not have the correct size, the TMP will populate the cache with new keys of the required size and write the cache data to the file so that they will be available the next time.

Currently, if two simulations are being run with TPM's that have different RSA key sizes (e.g,, one with 1024 and 2048 and another with 2048 and 3072, then the files will not match for the both of them and they will both try to overwrite the other's cache file. I may try to do something about this if necessary.

10.2.21.2 Includes, Types, Locals, and Defines

```
#include "Tpm.h"
#if USE_RSA_KEY_CACHE
#include  <stdio.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       privateExponent;
} RSA_KEY_CACHE;
```

Determine the number of RSA key sizes for the cache

```
TPMI_RSA_KEY_BITS       SupportedRsaKeySizes[] = {
#if RSA_1024
    1024,
#endif
#if RSA_2048
    2048,
#endif
#if RSA_3072
    3072,
#endif
#if RSA_4096
    4096,
#endif
    0
};

#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

```
32  RSA_KEY_CACHE        s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
33  // Indicates if the key cache is loaded. It can be loaded and enabled or disabled.
34  BOOL                 s_keyCacheLoaded = 0;
35
36  // Indicates if the key cache is enabled
37  int                  s_rsaKeyCacheEnabled = FALSE;
38
39  //*** RsaKeyCacheControl()
40  // Used to enable and disable the RSA key cache.
41  LIB_EXPORT void
42  RsaKeyCacheControl(
43      int             state
44      )
45  {
46      s_rsaKeyCacheEnabled = state;
47  }
```

10.2.21.2.1 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

```
48  static BOOL
49  InitializeKeyCache(
50      TPMT_PUBLIC         *publicArea,
51      TPMT_SENSITIVE      *sensitive,
52      RAND_STATE          *rand               // IN: if not NULL, the deterministic
53                                              //     RNG state
54      )
55  {
56      int                  index;
57      TPM_KEY_BITS         keySave = publicArea->parameters.rsaDetail.keyBits;
58      BOOL                 OK = TRUE;
59  //
60      s_rsaKeyCacheEnabled = FALSE;
61      for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
62      {
63          publicArea->parameters.rsaDetail.keyBits
64              = SupportedRsaKeySizes[index];
65          OK = (CryptRsaGenerateKey(publicArea, sensitive, rand) == TPM_RC_SUCCESS);
66          if(OK)
67          {
68              s_rsaKeyCache[index].publicModulus = publicArea->unique.rsa;
69              s_rsaKeyCache[index].privateExponent = sensitive->sensitive.rsa;
70          }
71      }
72      publicArea->parameters.rsaDetail.keyBits = keySave;
73      s_keyCacheLoaded = OK;
74  #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
75      if(OK)
76      {
77          FILE                *cacheFile;
78          const char          *fn = CACHE_FILE_NAME;
79
80  #if defined _MSC_VER
81          if(fopen_s(&cacheFile, fn, "w+b") != 0)
82  #else
```

```
83          cacheFile = fopen(fn, "w+b");
84          if(NULL == cacheFile)
85  #endif
86          {
87              printf("Can't open %s for write.\n", fn);
88          }
89          else
90          {
91              fseek(cacheFile, 0, SEEK_SET);
92              if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
93                 != sizeof(s_rsaKeyCache))
94              {
95                  printf("Error writing cache to %s.", fn);
96              }
97          }
98          if(cacheFile)
99              fclose(cacheFile);
100     }
101 #endif
102     return s_keyCacheLoaded;
103 }
```

10.2.21.2.2 KeyCacheLoaded()

Checks that key cache is loaded.

Return Value	Meaning
TRUE(1)	cache loaded
FALSE(0)	cache not loaded

```
104 static BOOL
105 KeyCacheLoaded(
106     TPMT_PUBLIC         *publicArea,
107     TPMT_SENSITIVE      *sensitive,
108     RAND_STATE          *rand               // IN: if not NULL, the deterministic
109                                             //     RNG state
110     )
111 {
112 #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
113     if(!s_keyCacheLoaded)
114     {
115         FILE            *cacheFile;
116         const char *     fn = CACHE_FILE_NAME;
117 #if defined _MSC_VER && 1
118         if(fopen_s(&cacheFile, fn, "r+b") == 0)
119 #else
120         cacheFile = fopen(fn, "r+b");
121         if(NULL != cacheFile)
122 #endif
123         {
124             fseek(cacheFile, 0L, SEEK_END);
125             if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
126             {
127                 fseek(cacheFile, 0L, SEEK_SET);
128                 s_keyCacheLoaded = (
129                     fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
130                     == sizeof(s_rsaKeyCache));
131             }
132             fclose(cacheFile);
133         }
134     }
135 #endif
```

```
136         if(!s_keyCacheLoaded)
137             s_rsaKeyCacheEnabled = InitializeKeyCache(publicArea, sensitive, rand);
138         return s_keyCacheLoaded;
139     }
```

10.2.21.2.3 GetCachedRsaKey()

Return Value	Meaning
TRUE(1)	key loaded
FALSE(0)	key not loaded

```
140     BOOL
141     GetCachedRsaKey(
142         TPMT_PUBLIC         *publicArea,
143         TPMT_SENSITIVE      *sensitive,
144         RAND_STATE          *rand               // IN: if not NULL, the deterministic
145                                                 //     RNG state
146         )
147     {
148         int                     keyBits = publicArea->parameters.rsaDetail.keyBits;
149         int                     index;
150     //
151         if(KeyCacheLoaded(publicArea, sensitive, rand))
152         {
153             for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
154             {
155                 if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
156                 {
157                     publicArea->unique.rsa = s_rsaKeyCache[index].publicModulus;
158                     sensitive->sensitive.rsa = s_rsaKeyCache[index].privateExponent;
159                     return TRUE;
160                 }
161             }
162             return FALSE;
163         }
164         return s_keyCacheLoaded;
165     }
166     #endif  // defined SIMULATION && defined USE_RSA_KEY_CACHE
```

10.2.22 Ticket.c

10.2.22.1 Introduction

This clause contains the functions used for ticket computations.

10.2.22.2 Includes

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

10.2.22.3 Functions

10.2.22.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE(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;
}
```

10.2.22.3.2 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  }
```

10.2.22.3.3 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 }
```

10.2.22.3.4 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 }
```

10.2.22.3.5 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     }
```

10.2.23 TpmAsn1.c

10.2.23.1 Includes

```
#include "Tpm.h"
#define _OIDS_
#include "OIDs.h"
#include "TpmASN1.h"
#include "TpmASN1_fp.h"
```

10.2.23.2 Unmarshaling Functions

10.2.23.2.1 ASN1UnmarshalContextInitialize()

Function does standard initialization of a context.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
BOOL
ASN1UnmarshalContextInitialize(
    ASN1UnmarshalContext    *ctx,
    INT16                    size,
    BYTE                    *buffer
)
{
    VERIFY(buffer != NULL);
    VERIFY(size > 0);
    ctx->buffer = buffer;
    ctx->size = size;
    ctx->offset = 0;
    ctx->tag = 0xFF;
    return TRUE;
Error:
    return FALSE;
}
```

10.2.23.2.2 ASN1DecodeLength()

This function extracts the length of an element from *buffer* starting at *offset*.

Return Value	Meaning
>=0	the extracted length
<0	an error

```
INT16
ASN1DecodeLength(
    ASN1UnmarshalContext        *ctx
)
{
    BYTE                first;                  // Next octet in buffer
    INT16               value;
//
    VERIFY(ctx->offset < ctx->size);
    first = NEXT_OCTET(ctx);
```

```
33      // If the number of octets of the entity is larger than 127, then the first octet
34      // is the number of octets in the length specifier.
35      if(first >= 0x80)
36      {
37          // Make sure that this length field is contained with the structure being
38          // parsed
39          CHECK_SIZE(ctx, (first & 0x7F));
40          if(first == 0x82)
41          {
42              // Two octets of size
43              // get the next value
44              value = (INT16)NEXT_OCTET(ctx);
45              // Make sure that the result will fit in an INT16
46              VERIFY(value < 0x0080);
47              // Shift up and add next octet
48              value = (value << 8) + NEXT_OCTET(ctx);
49          }
50          else if(first == 0x81)
51              value = NEXT_OCTET(ctx);
52          // Sizes larger than will fit in a INT16 are an error
53          else
54              goto Error;
55      }
56      else
57          value = first;
58      // Make sure that the size defined something within the current context
59      CHECK_SIZE(ctx, value);
60      return value;
61  Error:
62      ctx->size = -1;             // Makes everything fail from now on.
63      return -1;
64  }
```

10.2.23.2.3 ASN1NextTag()

This function extracts the next type from *buffer* starting at *offset*. It advances *offset* as it parses the type and the length of the type. It returns the length of the type. On return, the *length* octets starting at *offset* are the octets of the type.

Return Value	Meaning
>=0	the number of octets in *type*
<0	an error

```
65  INT16
66  ASN1NextTag(
67      ASN1UnmarshalContext    *ctx
68  )
69  {
70      // A tag to get?
71      VERIFY(ctx->offset < ctx->size);
72      // Get it
73      ctx->tag = NEXT_OCTET(ctx);
74      // Make sure that it is not an extended tag
75      VERIFY((ctx->tag & 0x1F) != 0x1F);
76      // Get the length field and return that
77      return ASN1DecodeLength(ctx);
78
79  Error:
80      // Attempt to read beyond the end of the context or an illegal tag
81      ctx->size = -1;         // Persistent failure
82      ctx->tag = 0xFF;
83      return -1;
```

```
84  }
```

10.2.23.2.4 ASN1GetBitStringValue()

Try to parse a bit string of up to 32 bits from a value that is expected to be a bit string. The bit string is left justified so that the MSb of the input is the MSb of the returned value. If there is a general parsing error, the context->size is set to -1.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
85  BOOL
86  ASN1GetBitStringValue(
87      ASN1UnmarshalContext        *ctx,
88      UINT32                      *val
89  )
90  {
91      int                 shift;
92      INT16               length;
93      UINT32              value = 0;
94      int                 inputBits;
95  //
96      length = ASN1NextTag(ctx);
97      VERIFY(length >= 1);
98      VERIFY(ctx->tag == ASN1_BITSTRING);
99      // Get the shift value for the bit field (how many bits to lop off of the end)
100     shift = NEXT_OCTET(ctx);
101     length--;
102     // Get the number of bits in the input
103     inputBits = (8 * length) - shift;
104     // the shift count has to make sense
105     VERIFY((shift < 8) && ((length > 0) || (shift == 0)));
106     // if there are any bytes left
107     for(; length > 1; length--)
108     {
109
110         // for all but the last octet, just shift and add the new octet
111         VERIFY((value & 0xFF000000) == 0); // can't loose significant bits
112         value = (value << 8) + NEXT_OCTET(ctx);
113
114     }
115     if(length == 1)
116     {
117         // for the last octet, just shift the accumulated value enough to
118         // accept the significant bits in the last octet and shift the last
119         // octet down
120         VERIFY(((value & (0xFF000000 << (8 - shift)))) == 0);
121         value = (value << (8 - shift)) + (NEXT_OCTET(ctx) >> shift);
122
123     }
124     // 'Left justify' the result
125     if(inputBits > 0)
126         value <<= (32 - inputBits);
127     *val = value;
128     return TRUE;
129 Error:
130     ctx->size = -1;
131     return FALSE;
132 }
```

10.2.23.3 Marshaling Functions

10.2.23.3.1 Introduction

Marshaling of an ASN.1 structure is accomplished from the bottom up. That is, the things that will be at the end of the structure are added last. To manage the collecting of the relative sizes, start a context for the outermost container, if there is one, and then placing items in from the bottom up. If the bottom-most item is also within a structure, create a nested context by calling ASN1StartMarshalingContext().

The context control structure contains a *buffer* pointer, an *offset*, an *end* and a stack. *offset* is the offset from the start of the buffer of the last added byte. When *offset* reaches 0, the buffer is full. *offset* is a signed value so that, when it becomes negative, there is an overflow. Only two functions are allowed to move bytes into the buffer: ASN1PushByte() and ASN1PushBytes(). These functions make sure that no data is written beyond the end of the buffer.

When a new context is started, the current value of *end* is pushed on the stack and *end* is set to 'offset. As bytes are added, offset gets smaller. At any time, the count of bytes in the current context is simply *end* - *offset*.

Since starting a new context involves setting *end* = *offset*, the number of bytes in the context starts at 0. The nominal way of ending a context is to use *end* - *offset* to set the length value, and then a tag is added to the buffer. Then the previous *end* value is popped meaning that the context just ended becomes a member of the now current context.

The nominal strategy for building a completed ASN.1 structure is to push everything into the buffer and then move everything to the start of the buffer. The move is simple as the size of the move is the initial *end* value minus the final *offset* value. The destination is *buffer* and the source is *buffer* + *offset*. As Skippy would say "Easy peasy, Joe."

It is not necessary to provide a buffer into which the data is placed. If no buffer is provided, then the marshaling process will return values needed for marshaling. On strategy for filling the buffer would be to execute the process for building the structure without using a buffer. This would return the overall size of the structure. Then that amount of data could be allocated for the buffer and the fill process executed again with the data going into the buffer. At the end, the data would be in its final resting place.

10.2.23.3.2 ASN1InitializeMarshalContext()

This creates a structure for handling marshaling of an ASN.1 formatted data structure.

```
133 void
134 ASN1InitializeMarshalContext(
135     ASN1MarshalContext      *ctx,
136     INT16                    length,
137     BYTE                    *buffer
138 )
139 {
140     ctx->buffer = buffer;
141     if(buffer)
142         ctx->offset = length;
143     else
144         ctx->offset = INT16_MAX;
145     ctx->end = ctx->offset;
146     ctx->depth = -1;
147 }
```

10.2.23.3.3 ASN1StartMarshalContext()

This starts a new constructed element. It is constructed on *top* of the value that was previously placed in the structure.

```
148  void
149  ASN1StartMarshalContext(
150      ASN1MarshalContext      *ctx
151  )
152  {
153      pAssert((ctx->depth + 1) < MAX_DEPTH);
154      ctx->depth++;
155      ctx->ends[ctx->depth] = ctx->end;
156      ctx->end = ctx->offset;
157  }
```

10.2.23.3.4 ASN1EndMarshalContext()

This function restores the end pointer for an encapsulating structure.

Return Value	Meaning
0	the size of the encapsulated structure that was just ended
0	an error

```
158  INT16
159  ASN1EndMarshalContext(
160      ASN1MarshalContext      *ctx
161  )
162  {
163      INT16                   length;
164      pAssert(ctx->depth >= 0);
165      length = ctx->end - ctx->offset;
166      ctx->end = ctx->ends[ctx->depth--];
167      if((ctx->depth == -1) && (ctx->buffer))
168      {
169          MemoryCopy(ctx->buffer, ctx->buffer + ctx->offset, ctx->end - ctx->offset);
170      }
171      return length;
172  }
```

10.2.23.3.5 ASN1EndEncapsulation()

This function puts a tag and length in the buffer. In this function, an embedded BIT_STRING is assumed to be a collection of octets. To indicate that all bits are used, a byte of zero is prepended. If a raw bit-string is needed, a new function like ASN1PushInteger() would be needed.

Return Value	Meaning
0	number of octets in the encapsulation
0	failure

```
173  UINT16
174  ASN1EndEncapsulation(
175      ASN1MarshalContext          *ctx,
176      BYTE                         tag
177  )
178  {
179      // only add a leading zero for an encapsulated BIT STRING
180      if (tag == ASN1_BITSTRING)
181          ASN1PushByte(ctx, 0);
182      ASN1PushTagAndLength(ctx, tag, ctx->end - ctx->offset);
183      return ASN1EndMarshalContext(ctx);
184  }
```

10.2.23.3.6 ASN1PushByte()

```
185  BOOL
186  ASN1PushByte(
187      ASN1MarshalContext          *ctx,
188      BYTE                         b
189  )
190  {
191      if(ctx->offset > 0)
192      {
193          ctx->offset -= 1;
194          if(ctx->buffer)
195              ctx->buffer[ctx->offset] = b;
196          return TRUE;
197      }
198      ctx->offset = -1;
199      return FALSE;
200  }
```

10.2.23.3.7 ASN1PushBytes()

Push some raw bytes onto the buffer. *count* cannot be zero.

Return Value	Meaning
0	count bytes
0	failure unless count was zero

```
201  INT16
202  ASN1PushBytes(
203      ASN1MarshalContext          *ctx,
204      INT16                        count,
205      const BYTE                  *buffer
206  )
207  {
208      // make sure that count is not negative which would mess up the math; and that
209      // if there is a count, there is a buffer
210      VERIFY((count >= 0) && ((buffer != NULL) || (count == 0)));
211      // back up the offset to determine where the new octets will get pushed
212      ctx->offset -= count;
213      // can't go negative
214      VERIFY(ctx->offset >= 0);
215      // if there are buffers, move the data, otherwise, assume that this is just a
216      // test.
217      if(count && buffer && ctx->buffer)
218          MemoryCopy(&ctx->buffer[ctx->offset], buffer, count);
219      return count;
220  Error:
221      ctx->offset = -1;
222      return 0;
223  }
```

10.2.23.3.8 ASN1PushNull()

Return Value	Meaning
0	count bytes
0	failure unless count was zero

```
224  INT16
```

```
225 ASN1PushNull(
226     ASN1MarshalContext      *ctx
227 )
228 {
229     ASN1PushByte(ctx, 0);
230     ASN1PushByte(ctx, ASN1_NULL);
231     return (ctx->offset >= 0) ? 2 : 0;
232 }
```

10.2.23.3.9 ASN1PushLength()

Push a length value. This will only handle length values that fit in an INT16.

Return Value	Meaning
0	number of bytes added
0	failure

```
233 INT16
234 ASN1PushLength(
235     ASN1MarshalContext          *ctx,
236     INT16                        len
237 )
238 {
239     UINT16                       start = ctx->offset;
240     VERIFY(len >= 0);
241     if(len <= 127)
242         ASN1PushByte(ctx, (BYTE)len);
243     else
244     {
245         ASN1PushByte(ctx, (BYTE)(len & 0xFF));
246         len >>= 8;
247         if(len == 0)
248             ASN1PushByte(ctx, 0x81);
249         else
250         {
251             ASN1PushByte(ctx, (BYTE)(len));
252             ASN1PushByte(ctx, 0x82);
253         }
254     }
255     goto Exit;
256 Error:
257     ctx->offset = -1;
258 Exit:
259     return (ctx->offset > 0) ? start - ctx->offset : 0;
260 }
```

10.2.23.3.10 ASN1PushTagAndLength()

Return Value	Meaning
0	number of bytes added
0	failure

```
261 INT16
262 ASN1PushTagAndLength(
263     ASN1MarshalContext          *ctx,
264     BYTE                         tag,
265     INT16                        length
266 )
267 {
```

```
268         INT16       bytes;
269         bytes = ASN1PushLength(ctx, length);
270         bytes += (INT16)ASN1PushByte(ctx, tag);
271         return (ctx->offset < 0) ? 0 : bytes;
272     }
```

10.2.23.3.11 ASN1PushTaggedOctetString()

This function will push a random octet string.

Return Value	Meaning
0	number of bytes added
0	failure

```
273     INT16
274     ASN1PushTaggedOctetString(
275         ASN1MarshalContext          *ctx,
276         INT16                        size,
277         const BYTE                  *string,
278         BYTE                         tag
279     )
280     {
281         ASN1PushBytes(ctx, size, string);
282         // PushTagAndLenght just tells how many octets it added so the total size of this
283         // element is the sum of those octets and input size.
284         size += ASN1PushTagAndLength(ctx, tag, size);
285         return size;
286     }
```

10.2.23.3.12 ASN1PushUINT()

This function pushes an native-endian integer value. This just changes a native-endian integer into a big-endian byte string and calls ASN1PushInteger(). That function will remove leading zeros and make sure that the number is positive.

Return Value	Meaning
0	count bytes
0	failure unless count was zero

```
287     INT16
288     ASN1PushUINT(
289         ASN1MarshalContext      *ctx,
290         UINT32                   integer
291     )
292     {
293         BYTE                    marshaled[4];
294         UINT32_TO_BYTE_ARRAY(integer, marshaled);
295         return ASN1PushInteger(ctx, 4, marshaled);
296     }
```

10.2.23.3.13 ASN1PushInteger

Push a big-endian integer on the end of the buffer

Return Value	Meaning
0	the number of bytes marshaled for the integer
0	failure

```
297   INT16
298   ASN1PushInteger(
299       ASN1MarshalContext  *ctx,           // IN/OUT: buffer context
300       INT16                iLen,          // IN: octets of the integer
301       BYTE                *integer        // IN: big-endian integer
302   )
303   {
304       // no leading 0's
305       while((*integer == 0) && (--iLen > 0))
306           integer++;
307       // Move the bytes to the buffer
308       ASN1PushBytes(ctx, iLen, integer);
309       // if needed, add a leading byte of 0 to make the number positive
310       if(*integer & 0x80)
311           iLen += (INT16)ASN1PushByte(ctx, 0);
312       // PushTagAndLenght just tells how many octets it added so the total size of this
313       // element is the sum of those octets and the adjusted input size.
314       iLen +=  ASN1PushTagAndLength(ctx, ASN1_INTEGER, iLen);
315       return iLen;
316   }
```

10.2.23.3.14 ASN1PushOID()

This function is used to add an OID. An OID is 0x06 followed by a byte of size followed by size bytes. This is used to avoid having to do anything special in the definition of an OID.

Return Value	Meaning
0	the number of bytes marshaled for the integer
0	failure

```
317   INT16
318   ASN1PushOID(
319       ASN1MarshalContext          *ctx,
320       const BYTE                  *OID
321   )
322   {
323       if((*OID == ASN1_OBJECT_IDENTIFIER) && ((OID[1] & 0x80) == 0))
324       {
325           return ASN1PushBytes(ctx, OID[1] + 2, OID);
326       }
327       ctx->offset = -1;
328       return 0;
329   }
```

10.2.24 X509_ECC.c

10.2.24.1 Includes

```
#include "Tpm.h"
#include "X509.h"
#include "OIDs.h"
#include "TpmASN1_fp.h"
#include "X509_spt_fp.h"
#include "CryptHash_fp.h"
```

10.2.24.2 Functions

10.2.24.2.1 X509PushPoint()

This seems like it might be used more than once so...

Return Value	Meaning
0	number of bytes added
0	failure

```
INT16
X509PushPoint(
    ASN1MarshalContext      *ctx,
    TPMS_ECC_POINT          *p
)
{
    // Push a bit string containing the public key. For now, push the x, and y
    // coordinates of the public point, bottom up
    ASN1StartMarshalContext(ctx); // BIT STRING
    {
        ASN1PushBytes(ctx, p->y.t.size, p->y.t.buffer);
        ASN1PushBytes(ctx, p->x.t.size, p->x.t.buffer);
        ASN1PushByte(ctx, 0x04);
    }
    return ASN1EndEncapsulation(ctx, ASN1_BITSTRING); // Ends BIT STRING
}
```

10.2.24.2.2 X509AddSigningAlgorithmECC()

This creates the singing algorithm data.

Return Value	Meaning
0	number of bytes added
0	failure

```
INT16
X509AddSigningAlgorithmECC(
    OBJECT              *signKey,
    TPMT_SIG_SCHEME     *scheme,
    ASN1MarshalContext  *ctx
)
{
    PHASH_DEF            hashDef = CryptGetHashDef(scheme->details.any.hashAlg);
//
    NOT_REFERENCED(signKey);
```

```
33      // If the desired hashAlg definition wasn't found...
34      if(hashDef->hashAlg != scheme->details.any.hashAlg)
35          return 0;
36
37      switch(scheme->scheme)
38      {
39          case ALG_ECDSA_VALUE:
40              // Make sure that we have an OID for this hash and ECC
41              if((hashDef->ECDSA)[0] != ASN1_OBJECT_IDENTIFIER)
42                  break;
43              // if this is just an implementation check, indicate that this
44              // combination is supported
45              if(!ctx)
46                  return 1;
47              ASN1StartMarshalContext(ctx);
48              ASN1PushOID(ctx, hashDef->ECDSA);
49              return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
50          default:
51              break;
52      }
53      return 0;
54  }
```

10.2.24.2.3 X509AddPublicECC()

This function will add the *publicKey* description to the DER data. If ctx is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
0	number of bytes added
0	failure

```
55  INT16
56  X509AddPublicECC(
57      OBJECT                  *object,
58      ASN1MarshalContext      *ctx
59  )
60  {
61      const BYTE      *curveOid =
62          CryptEccGetOID(object->publicArea.parameters.eccDetail.curveID);
63      if((curveOid == NULL) || (*curveOid != ASN1_OBJECT_IDENTIFIER))
64          return 0;
65  //
66  //
67  //  SEQUENCE (2 elem) 1st
68  //    SEQUENCE (2 elem) 2nd
69  //      OBJECT IDENTIFIER 1.2.840.10045.2.1 ecPublicKey (ANSI X9.62 public key type)
70  //      OBJECT IDENTIFIER 1.2.840.10045.3.1.7 prime256v1 (ANSI X9.62 named curve)
71  //    BIT STRING (520 bit) 000001001010000111010101010111001001101101000100000010...
72  //
73      // If this is a check to see if the key can be encoded, it can.
74      // Need to mark the end sequence
75      if(ctx == NULL)
76          return 1;
77      ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
78      {
79          X509PushPoint(ctx, &object->publicArea.unique.ecc); // BIT STRING
80          ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 2nd
81          {
82              ASN1PushOID(ctx, curveOid); // curve dependent
83              ASN1PushOID(ctx, OID_ECC_PUBLIC); // (1.2.840.10045.2.1)
84          }
```

```
85              ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 2nd
86          }
87          return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 1st
88      }
```

10.2.25 X509_RSA.c

10.2.25.1 Includes

```
#include "Tpm.h"
#include "X509.h"
#include "TpmASN1_fp.h"
#include "X509_spt_fp.h"
#include "CryptHash_fp.h"
#include "CryptRsa_fp.h"
```

10.2.25.2 Functions

```
#if ALG_RSA
```

10.2.25.2.1 X509AddSigningAlgorithmRSA()

This creates the singing algorithm data.

Return Value	Meaning
0	number of bytes added
0	failure

```
INT16
X509AddSigningAlgorithmRSA(
    OBJECT              *signKey,
    TPMT_SIG_SCHEME     *scheme,
    ASN1MarshalContext  *ctx
)
{
    TPM_ALG_ID           hashAlg = scheme->details.any.hashAlg;
    PHASH_DEF            hashDef = CryptGetHashDef(hashAlg);
//
    NOT_REFERENCED(signKey);
    // return failure if hash isn't implemented
    if(hashDef->hashAlg != hashAlg)
        return 0;
    switch(scheme->scheme)
    {
        case ALG_RSASSA_VALUE:
        {
            // if the hash is implemented but there is no PKCS1 OID defined
            // then this is not a valid signing combination.
            if(hashDef->PKCS1[0] != ASN1_OBJECT_IDENTIFIER)
                break;
            if(ctx == NULL)
                return 1;
            return X509PushAlgorithmIdentifierSequence(ctx, hashDef->PKCS1);
        }
        case ALG_RSAPSS_VALUE:
            // leave if this is just an implementation check
            if(ctx == NULL)
                return 1;
            // In the case of SHA1, everything is default and RFC4055 says that
            // implementations that do signature generation MUST omit the parameter
            // when defaults are used. )-:
            if(hashDef->hashAlg == ALG_SHA1_VALUE)
            {
                return X509PushAlgorithmIdentifierSequence(ctx, OID_RSAPSS);
            }
```

```
45              else
46              {
47                  // Going to build something that looks like:
48                  //  SEQUENCE (2 elem)
49                  //     OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
50                  //     SEQUENCE (3 elem)
51                  //       [0] (1 elem)
52                  //         SEQUENCE (2 elem)
53                  //           OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
54                  //           NULL
55                  //       [1] (1 elem)
56                  //         SEQUENCE (2 elem)
57                  //           OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
58                  //           SEQUENCE (2 elem)
59                  //             OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
60                  //             NULL
61                  //       [2] (1 elem)  salt length
62                  //         INTEGER 32
63
64                  // The indentation is just to keep track of where we are in the
65                  // structure
66                  ASN1StartMarshalContext(ctx); // SEQUENCE (2 elements)
67                  {
68                      ASN1StartMarshalContext(ctx);   // SEQUENCE (3 elements)
69                      {
70                          // [2] (1 elem)  salt length
71                          //    INTEGER 32
72                          ASN1StartMarshalContext(ctx);
73                          {
74                              INT16       saltSize =
75                                  CryptRsaPssSaltSize((INT16)hashDef->digestSize,
76                                  (INT16)signKey->publicArea.unique.rsa.t.size);
77                              ASN1PushUINT(ctx, saltSize);
78                          }
79                          ASN1EndEncapsulation(ctx, ASN1_APPLICAIION_SPECIFIC + 2);
80
81                          // Add the mask generation algorithm
82                          // [1] (1 elem)
83                          //    SEQUENCE (2 elem) 1st
84                          //      OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
85                          //      SEQUENCE (2 elem) 2nd
86                          //        OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
87                          //        NULL
88                          ASN1StartMarshalContext(ctx);   // mask context [1] (1 elem)
89                          {
90                              ASN1StartMarshalContext(ctx);   // SEQUENCE (2 elem) 1st
91                              // Handle the 2nd Sequence (sequence (object, null))
92                              {
93                                  // This adds a NULL, then an OID and a SEQUENCE
94                                  // wrapper.
95                                  X509PushAlgorithmIdentifierSequence(ctx,
96                                      hashDef->OID);
97                                  // add the pkcs1-MGF OID
98                                  ASN1PushOID(ctx, OID_MGF1);
99                              }
100                             // End outer sequence
101                             ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
102                         }
103                         // End the [1]
104                         ASN1EndEncapsulation(ctx, ASN1_APPLICAIION_SPECIFIC + 1);
105
106                         // Add the hash algorithm
107                         // [0] (1 elem)
108                         //   SEQUENCE (2 elem) (done by
109                         //               X509PushAlgorithmIdentifierSequence)
110                         //     OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256 (NIST)
```

```
                        //     NULL
                        ASN1StartMarshalContext(ctx); // [0] (1 elem)
                        {
                            X509PushAlgorithmIdentifierSequence(ctx, hashDef->OID);
                        }
                        ASN1EndEncapsulation(ctx, (ASN1_APPLICAIION_SPECIFIC + 0));
                    }
                    //  SEQUENCE (3 elements) end
                    ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);

                    // RSA PSS OID
                    // OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
                    ASN1PushOID(ctx, OID_RSAPSS);
                }
                // End Sequence (2 elements)
                return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
            }
        default:
            break;
    }
    return 0;
}
```

10.2.25.2.2 X509AddPublicRSA()

This function will add the *publicKey* description to the DER data. If *fillPtr* is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
0	number of bytes added
0	failure

```
INT16
X509AddPublicRSA(
    OBJECT                  *object,
    ASN1MarshalContext    *ctx
)
{
    UINT32          exp = object->publicArea.parameters.rsaDetail.exponent;
//
/*
    SEQUENCE (2 elem) 1st
      SEQUENCE (2 elem) 2nd
        OBJECT IDENTIFIER 1.2.840.113549.1.1.1 rsaEncryption (PKCS #1)
        NULL
      BIT STRING (1 elem)
        SEQUENCE (2 elem) 3rd
          INTEGER (2048 bit) 21973045137412279557258341993574O1
          INTEGER 65537
*/
    // If this is a check to see if the key can be encoded, it can.
    // Need to mark the end sequence
    if(ctx == NULL)
        return 1;
    ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
    ASN1StartMarshalContext(ctx); // BIT STRING
    ASN1StartMarshalContext(ctx); // SEQUENCE *(2 elem) 3rd

    // Get public exponent in big-endian byte order.
    if(exp == 0)
        exp = RSA_DEFAULT_PUBLIC_EXPONENT;

```

```
163         // Push a 4 byte integer. This might get reduced if there are leading zeros or
164         // extended if the high order byte is negative.
165         ASN1PushUINT(ctx, exp);
166         // Push the public key as an integer
167         ASN1PushInteger(ctx, object->publicArea.unique.rsa.t.size,
168                                 object->publicArea.unique.rsa.t.buffer);
169         // Embed this in a SEQUENCE tag and length in for the key, exponent sequence
170         ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // SEQUENCE (3rd)
171
172         // Embed this in a BIT STRING
173         ASN1EndEncapsulation(ctx, ASN1_BITSTRING);
174
175         // Now add the formatted SEQUENCE for the RSA public key OID. This is a
176         // fully constructed value so it doesn't need to have a context started
177         X509PushAlgorithmIdentifierSequence(ctx, OID_PKCS1_PUB);
178
179         return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
180     }
181     #endif // ALG_RSA
```

10.2.26 X509_spt.c

10.2.26.1 Includes

```
#include "Tpm.h"
#include "TpmASN1.h"
#include "TpmASN1_fp.h"
#define _X509_SPT_
#include "X509.h"
#include "X509_spt_fp.h"
#if ALG_RSA
#   include "X509_RSA_fp.h"
#endif // ALG_RSA
#if ALG_ECC
#   include "X509_ECC_fp.h"
#endif // ALG_ECC
#if ALG_SM2
//#   include "X509_SM2_fp.h"
#endif // ALG_RSA
```

10.2.26.2 Unmarshaling Functions

10.2.26.2.1 X509FindExtensionByOID()

This will search a list of X509 extensions to find an extension with the requested OID. If the extension is found, the output context (*ctx*) is set up to point to the OID in the extension.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure (could be catastrophic)

```
BOOL
X509FindExtensionByOID(
    ASN1UnmarshalContext    *ctxIn,         // IN: the context to search
    ASN1UnmarshalContext    *ctx,           // OUT: the extension context
    const BYTE              *OID            // IN: oid to search for
)
{
    INT16                 length;
//
    pAssert(ctxIn != NULL);
    // Make the search non-destructive of the input if ctx provided. Otherwise, use
    // the provided context.
    if (ctx == NULL)
        ctx = ctxIn;
    // if the provide search context is different from the context of the extension,
    // then copy the search context to the search context.
    else if(ctx != ctxIn)
        *ctx = *ctxIn;
    // Now, search in the extension context
    for(;ctx->size > ctx->offset; ctx->offset += length)
    {
        VERIFY((length = ASN1NextTag(ctx)) >= 0);
        // If this is not a constructed sequence, then it doesn't belong
        // in the extensions.
        VERIFY(ctx->tag == ASN1_CONSTRUCTED_SEQUENCE);
        // Make sure that this entry could hold the OID
        if (length >= OID_SIZE(OID))
        {
            // See if this is a match for the provided object identifier.
```

```
45              if (MemoryEqual(OID, &(ctx->buffer[ctx->offset]), OID_SIZE(OID)))
46              {
47                  // Return with ' ctx' set to point to the start of the OID with the
    size
48                  // set to be the size of the SEQUENCE
49                  ctx->buffer += ctx->offset;
50                  ctx->offset = 0;
51                  ctx->size = length;
52                  return TRUE;
53              }
54          }
55      }
56      VERIFY(ctx->offset == ctx->size);
57      return FALSE;
58  Error:
59      ctxIn->size = -1;
60      ctx->size = -1;
61      return FALSE;
62  }
```

10.2.26.2.2 X509GetExtensionBits()

This function will extract a bit field from an extension. If the extension doesn't contain a bit string, it will fail.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
63  UINT32
64  X509GetExtensionBits(
65      ASN1UnmarshalContext            *ctx,
66      UINT32                          *value
67  )
68  {
69      INT16                 length;
70  //
71      while (((length = ASN1NextTag(ctx)) > 0) && (ctx->size > ctx->offset))
72      {
73          // Since this is an extension, the extension value will be in an OCTET STRING
74          if (ctx->tag == ASN1_OCTET_STRING)
75          {
76              return ASN1GetBitStringValue(ctx, value);
77          }
78          ctx->offset += length;
79      }
80      ctx->size = -1;
81      return FALSE;
82  }
```

10.2.26.2.3 X509ProcessExtensions()

This function is used to process the TPMA_OBJECT and KeyUsage() extensions. It is not in the CertifyX509.c code because it makes the code harder to follow.

Error Returns	Meaning
TPM_RCS_ATTRIBUTES	the attributes of object are not consistent with the extension setting
TPM_RC_VALUE	problem parsing the extensions

```
TPM_RC
X509ProcessExtensions(
    OBJECT              *object,        // IN: The object with the attributes to
                                        //      check
    stringRef           *extension      // IN: The start and length of the extensions
)
{
    ASN1UnmarshalContext     ctx;
    ASN1UnmarshalContext     extensionCtx;
    INT16                    length;
    UINT32                   value;
    TPMA_OBJECT              attributes = object->publicArea.objectAttributes;
//
    if(!ASN1UnmarshalContextInitialize(&ctx, extension->len, extension->buf)
       || ((length = ASN1NextTag(&ctx)) < 0)
       || (ctx.tag != X509_EXTENSIONS))
        return TPM_RCS_VALUE;
    if( ((length = ASN1NextTag(&ctx)) < 0)
       || (ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE)))
        return TPM_RCS_VALUE;

    // Get the extension for the TPMA_OBJECT if there is one
    if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_TCG_TPMA_OBJECT) &&
        X509GetExtensionBits(&extensionCtx, &value))
    {
        // If an keyAttributes extension was found, it must be exactly the same as the
        // attributes of the object.
        // NOTE: MemoryEqual() is used rather than a simple UINT32 compare to avoid
        // type-punned pointer warning/error.
        if(!MemoryEqual(&value, &attributes, sizeof(value)))
            return TPM_RCS_ATTRIBUTES;
    }
    // Make sure the failure to find the value wasn't because of a fatal error
    else if(extensionCtx.size < 0)
        return TPM_RCS_VALUE;

    // Get the keyUsage extension. This one is required
    if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_KEY_USAGE_EXTENSION) &&
        X509GetExtensionBits(&extensionCtx, &value))
    {
        x509KeyUsageUnion   keyUsage;
        BOOL                bad;
    //
        keyUsage.integer = value;
        // For KeyUsage:
        // 1) 'sign' is SET if Key Usage includes signing
        bad = (KEY_USAGE_SIGN.integer & keyUsage.integer) != 0
              && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
        // 2) 'decrypt' is SET if Key Usage includes decryption uses
        bad = bad || (KEY_USAGE_DECRYPT.integer & keyUsage.integer) != 0
                  && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
        // 3) 'fixedTPM' is SET if Key Usage is non-repudiation
        bad = bad || IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, nonrepudiation)
                   && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM);
        // 4)'restricted' is SET if Key Usage is for key agreement.
        bad = bad || IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, keyAgreement)
                  && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted);
        if(bad)
            return TPM_RCS_VALUE;
```

```
142         }
143         else
144             // The KeyUsage extension is required
145             return TPM_RCS_VALUE;
146 
147         return TPM_RC_SUCCESS;
148     }
```

10.2.26.3 Marshaling Functions

10.2.26.3.1 X509AddSigningAlgorithm()

This creates the singing algorithm data.

Return Value	Meaning
0	number of octets added
0	failure

```
149     INT16
150     X509AddSigningAlgorithm(
151         ASN1MarshalContext  *ctx,
152         OBJECT              *signKey,
153         TPMT_SIG_SCHEME     *scheme
154     )
155     {
156         switch(signKey->publicArea.type)
157         {
158     #if ALG_RSA
159             case ALG_RSA_VALUE:
160                 return X509AddSigningAlgorithmRSA(signKey, scheme, ctx);
161     #endif // ALG_RSA
162     #if ALG_ECC
163             case ALG_ECC_VALUE:
164                 return X509AddSigningAlgorithmECC(signKey, scheme, ctx);
165     #endif // ALG_ECC
166     #if ALG_SM2
167             case ALG_SM2:
168                 return X509AddSigningAlgorithmSM2(signKey, scheme,ctx);
169     #endif // ALG_SM2
170             default:
171                 break;
172         }
173         return 0;
174     }
```

10.2.26.3.2 X509AddPublicKey()

This function will add the *publicKey* description to the DER data. If *fillPtr* is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
0	number of octets added
0	failure

```
175     INT16
176     X509AddPublicKey(
177         ASN1MarshalContext  *ctx,
178         OBJECT              *object
```

```
179 )
180 {
181     switch(object->publicArea.type)
182     {
183 #if ALG_RSA
184         case ALG_RSA_VALUE:
185             return X509AddPublicRSA(object, ctx);
186 #endif
187 #if ALG_ECC
188         case ALG_ECC_VALUE:
189             return X509AddPublicECC(object, ctx);
190 #endif
191 #if ALG_SM2
192         case ALG_SM2_VALUE:
193             break;
194 #endif
195         default:
196             break;
197     }
198     return FALSE;
199 }
```

10.2.26.3.3 X509PushAlgorithmIdentifierSequence()

The function adds the algorithm identifier sequence.

Return Value	Meaning
0	number of bytes added
0	failure

```
200 INT16
201 X509PushAlgorithmIdentifierSequence(
202     ASN1MarshalContext          *ctx,
203     const BYTE                  *OID
204     )
205 {
206     ASN1StartMarshalContext(ctx);   // hash algorithm
207     ASN1PushNull(ctx);
208     ASN1PushOID(ctx, OID);
209     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
210 }
```

10.2.27 AC_spt.c

10.2.27.1 Includes

```
#include "Tpm.h"
#include "AC_spt_fp.h"
#if 1 // This is the simulated AC data.
typedef struct {
    TPMI_RH_AC                ac;
    TPML_AC_CAPABILITIES    *acData;

} acCapabilities;
TPML_AC_CAPABILITIES acData0001 = {1,
        {{TPM_AT_PV1, 0x01234567}}};

acCapabilities  ac[1] = { {0x0001, &acData0001} };

#define NUM_AC  (sizeof(ac) / sizeof(acCapabilities))
#endif // 1 The simulated AC data
```

10.2.27.1.1 AcToCapabilities()

This function returns a pointer to a list of AC capabilities.

```
TPML_AC_CAPABILITIES *
AcToCapabilities(
    TPMI_RH_AC      component       // IN: component
)
{
    UINT32          index;
//
    for(index = 0; index < NUM_AC; index++)
    {
        if(ac[index].ac == component)
            return ac[index].acData;
    }
    return NULL;
}
```

10.2.27.1.2 AcIsAccessible()

Function to determine if an AC handle references an actual AC

Return Value	Meaning

```
BOOL
AcIsAccessible(
    TPM_HANDLE          acHandle
    )
{
    // In this implementation, the AC exists if there are some capabilities to go
    // with the handle
    return AcToCapabilities(acHandle) != NULL;
}
```

10.2.27.1.3 AcCapabilitiesGet()

This function returns a list of capabilities associated with an AC

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

```
39  TPMI_YES_NO
40  AcCapabilitiesGet(
41      TPMI_RH_AC               component,     // IN: the component
42      TPM_AT                   type,          // IN: start capability type
43      TPML_AC_CAPABILITIES    *capabilityList // OUT: list of handle
44  )
45  {
46      TPMI_YES_NO              more = NO;
47      UINT32                   i;
48      TPML_AC_CAPABILITIES    *capabilities = AcToCapabilities(component);
49
50      pAssert(HandleGetType(component) == TPM_HT_AC);
51
52      // Initialize output handle list
53      capabilityList->count = 0;
54
55      if(capabilities != NULL)
56      {
57          // Find the first capability less than or equal to type
58          for(i = 0; i < capabilities->count; i++)
59          {
60              if(capabilities->acCapabilities[i].tag >= type)
61              {
62                  // copy the capabilities until we run out or fill the list
63                  for(; (capabilityList->count < MAX_AC_CAPABILITIES)
64                      && (i < capabilities->count); i++)
65                  {
66                      capabilityList->acCapabilities[capabilityList->count]
67                          = capabilities->acCapabilities[i];
68                      capabilityList->count++;
69                  }
70                  more = i < capabilities->count;
71              }
72          }
73      }
74      return more;
75  }
```

10.2.27.1.4 AcSendObject()

Stub to handle sending of an AC object

Error Returns	Meaning

```
76  TPM_RC
77  AcSendObject(
78      TPM_HANDLE           acHandle,      // IN: Handle of AC receiving object
79      OBJECT              *object,        // IN: object structure to send
80      TPMS_AC_OUTPUT      *acDataOut      // OUT: results of operation
81  )
82  {
83      NOT_REFERENCED(object);
84      NOT_REFERENCED(acHandle);
85      acDataOut->tag = TPM_AT_ERROR;  // indicate that the response contains an
86                                      // error code
87      acDataOut->data = TPM_AE_NONE;  // but there is no error.
88
```

```
89          return TPM_RC_SUCCESS;
90      }
```

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are used to define a TPM profile. The values are 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:

1) The TCG_Algorithm Registery,
2) Part 2 of this specification, and
3) A purpose-built document that contains vendor-specific information in tables.

All of the values in this file have #ifdef 'guards' so that they may be defined in a command line.Additionally, TpmBuildSwitches.h allows an additional file to be specified in the compiler command line and preset any of these values.

A.2 TpmProfile.h

```
#ifndef _TPM_PROFILE_H_
#define _TPM_PROFILE_H_
```

Table 2:4 - Defines for Logic Values

```
#undef TRUE
#define TRUE                1
#undef FALSE
#define FALSE               0
#undef YES
#define YES                 1
#undef NO
#define NO                  0
#undef SET
#define SET                 1
#undef CLEAR
#define CLEAR               0
```

Table 0:1 - Defines for Processor Values

```
#ifndef BIG_ENDIAN_TPM
#define BIG_ENDIAN_TPM              NO
#endif
#ifndef LITTLE_ENDIAN_TPM
#define LITTLE_ENDIAN_TPM           !BIG_ENDIAN_TPM
#endif
#ifndef MOST_SIGNIFICANT_BIT_0
#define MOST_SIGNIFICANT_BIT_0      NO
#endif
#ifndef LEAST_SIGNIFICANT_BIT_0
#define LEAST_SIGNIFICANT_BIT_0     !MOST_SIGNIFICANT_BIT_0
#endif
#ifndef AUTO_ALIGN
#define AUTO_ALIGN                  NO
#endif
```

Table 0:4 - Defines for Implemented Curves

```
30  #ifndef ECC_NIST_P192
31  #define ECC_NIST_P192                   NO
32  #endif
33  #ifndef ECC_NIST_P224
34  #define ECC_NIST_P224                   NO
35  #endif
36  #ifndef ECC_NIST_P256
37  #define ECC_NIST_P256                   YES
38  #endif
39  #ifndef ECC_NIST_P384
40  #define ECC_NIST_P384                   YES
41  #endif
42  #ifndef ECC_NIST_P521
43  #define ECC_NIST_P521                   NO
44  #endif
45  #ifndef ECC_BN_P256
46  #define ECC_BN_P256                     YES
47  #endif
48  #ifndef ECC_BN_P638
49  #define ECC_BN_P638                     NO
50  #endif
51  #ifndef ECC_SM2_P256
52  #define ECC_SM2_P256                    NO
53  #endif
```

Table 0:6 - Defines for Implemented ACT

```
54  #ifndef RH_ACT_0
55  #define RH_ACT_0                        YES
56  #endif
57  #ifndef RH_ACT_1
58  #define RH_ACT_1                        NO
59  #endif
60  #ifndef RH_ACT_A
61  #define RH_ACT_A                        YES
62  #endif
```

Table 0:7 - Defines for Implementation Values

```
63  #ifndef FIELD_UPGRADE_IMPLEMENTED
64  #define FIELD_UPGRADE_IMPLEMENTED       NO
65  #endif
66  #ifndef HASH_LIB
67  #define HASH_LIB                        Ossl
68  #endif
69  #ifndef SYM_LIB
70  #define SYM_LIB                         Ossl
71  #endif
72  #ifndef MATH_LIB
73  #define MATH_LIB                        Ossl
74  #endif
75  #ifndef IMPLEMENTATION_PCR
76  #define IMPLEMENTATION_PCR              24
77  #endif
78  #ifndef PLATFORM_PCR
79  #define PLATFORM_PCR                    24
80  #endif
81  #ifndef DRTM_PCR
82  #define DRTM_PCR                        17
83  #endif
84  #ifndef HCRTM_PCR
85  #define HCRTM_PCR                       0
86  #endif
87  #ifndef NUM_LOCALITIES
88  #define NUM_LOCALITIES                  5
```

```
#endif
#ifndef MAX_HANDLE_NUM
#define MAX_HANDLE_NUM                  3
#endif
#ifndef MAX_ACTIVE_SESSIONS
#define MAX_ACTIVE_SESSIONS             64
#endif
#ifndef CONTEXT_SLOT
#define CONTEXT_SLOT                    UINT16
#endif
#ifndef MAX_LOADED_SESSIONS
#define MAX_LOADED_SESSIONS             3
#endif
#ifndef MAX_SESSION_NUM
#define MAX_SESSION_NUM                 3
#endif
#ifndef MAX_LOADED_OBJECTS
#define MAX_LOADED_OBJECTS              3
#endif
#ifndef MIN_EVICT_OBJECTS
#define MIN_EVICT_OBJECTS               2
#endif
#ifndef NUM_POLICY_PCR_GROUP
#define NUM_POLICY_PCR_GROUP            1
#endif
#ifndef NUM_AUTHVALUE_PCR_GROUP
#define NUM_AUTHVALUE_PCR_GROUP         1
#endif
#ifndef MAX_CONTEXT_SIZE
#define MAX_CONTEXT_SIZE                1264
#endif
#ifndef MAX_DIGEST_BUFFER
#define MAX_DIGEST_BUFFER               1024
#endif
#ifndef MAX_NV_INDEX_SIZE
#define MAX_NV_INDEX_SIZE               2048
#endif
#ifndef MAX_NV_BUFFER_SIZE
#define MAX_NV_BUFFER_SIZE              1024
#endif
#ifndef MAX_CAP_BUFFER
#define MAX_CAP_BUFFER                  1024
#endif
#ifndef NV_MEMORY_SIZE
#define NV_MEMORY_SIZE                  16384
#endif
#ifndef MIN_COUNTER_INDICES
#define MIN_COUNTER_INDICES             8
#endif
#ifndef NUM_STATIC_PCR
#define NUM_STATIC_PCR                  16
#endif
#ifndef MAX_ALG_LIST_SIZE
#define MAX_ALG_LIST_SIZE               64
#endif
#ifndef PRIMARY_SEED_SIZE
#define PRIMARY_SEED_SIZE               32
#endif
#ifndef CONTEXT_ENCRYPT_ALGORITHM
#define CONTEXT_ENCRYPT_ALGORITHM       AES
#endif
#ifndef NV_CLOCK_UPDATE_INTERVAL
#define NV_CLOCK_UPDATE_INTERVAL        12
#endif
#ifndef NUM_POLICY_PCR
#define NUM_POLICY_PCR                  1
```

```
155   #endif
156   #ifndef MAX_COMMAND_SIZE
157   #define MAX_COMMAND_SIZE                4096
158   #endif
159   #ifndef MAX_RESPONSE_SIZE
160   #define MAX_RESPONSE_SIZE               4096
161   #endif
162   #ifndef ORDERLY_BITS
163   #define ORDERLY_BITS                    8
164   #endif
165   #ifndef MAX_SYM_DATA
166   #define MAX_SYM_DATA                    128
167   #endif
168   #ifndef MAX_RNG_ENTROPY_SIZE
169   #define MAX_RNG_ENTROPY_SIZE            64
170   #endif
171   #ifndef RAM_INDEX_SPACE
172   #define RAM_INDEX_SPACE                 512
173   #endif
174   #ifndef RSA_DEFAULT_PUBLIC_EXPONENT
175   #define RSA_DEFAULT_PUBLIC_EXPONENT     0x00010001
176   #endif
177   #ifndef ENABLE_PCR_NO_INCREMENT
178   #define ENABLE_PCR_NO_INCREMENT         YES
179   #endif
180   #ifndef CRT_FORMAT_RSA
181   #define CRT_FORMAT_RSA                  YES
182   #endif
183   #ifndef VENDOR_COMMAND_COUNT
184   #define VENDOR_COMMAND_COUNT            0
185   #endif
186   #ifndef MAX_VENDOR_BUFFER_SIZE
187   #define MAX_VENDOR_BUFFER_SIZE          1024
188   #endif
189   #ifndef MAX_DERIVATION_BITS
190   #define MAX_DERIVATION_BITS             8192
191   #endif
192   #ifndef SIZE_OF_X509_SERIAL_NUMBER
193   #define SIZE_OF_X509_SERIAL_NUMBER      20
194   #endif
195   #ifndef PRIVATE_VENDOR_SPECIFIC_BYTES
196   #define PRIVATE_VENDOR_SPECIFIC_BYTES   RSA_PRIVATE_SIZE
197   #endif
```

Table 0:2 - Defines for Implemented Algorithms

```
198   #ifndef ALG_AES
199   #define ALG_AES                         ALG_YES
200   #endif
201   #ifndef ALG_CAMELLIA
202   #define ALG_CAMELLIA                    ALG_YES
203   #endif
204   #ifndef ALG_CBC
205   #define ALG_CBC                         ALG_YES
206   #endif
207   #ifndef ALG_CFB
208   #define ALG_CFB                         ALG_YES
209   #endif
210   #ifndef ALG_CMAC
211   #define ALG_CMAC                        ALG_YES
212   #endif
213   #ifndef ALG_CTR
214   #define ALG_CTR                         ALG_YES
215   #endif
216   #ifndef ALG_ECB
```

```
217   #define ALG_ECB                     ALG_YES
218   #endif
219   #ifndef ALG_ECC
220   #define ALG_ECC                     ALG_YES
221   #endif
222   #ifndef ALG_ECDAA
223   #define ALG_ECDAA                   (ALG_YES && ALG_ECC)
224   #endif
225   #ifndef ALG_ECDH
226   #define ALG_ECDH                    (ALG_YES && ALG_ECC)
227   #endif
228   #ifndef ALG_ECDSA
229   #define ALG_ECDSA                   (ALG_YES && ALG_ECC)
230   #endif
231   #ifndef ALG_ECMQV
232   #define ALG_ECMQV                   (ALG_NO && ALG_ECC)
233   #endif
234   #ifndef ALG_ECSCHNORR
235   #define ALG_ECSCHNORR               (ALG_YES && ALG_ECC)
236   #endif
237   #ifndef ALG_HMAC
238   #define ALG_HMAC                    ALG_YES
239   #endif
240   #ifndef ALG_KDF1_SP800_108
241   #define ALG_KDF1_SP800_108          ALG_YES
242   #endif
243   #ifndef ALG_KDF1_SP800_56A
244   #define ALG_KDF1_SP800_56A          (ALG_YES && ALG_ECC)
245   #endif
246   #ifndef ALG_KDF2
247   #define ALG_KDF2                    ALG_NO
248   #endif
249   #ifndef ALG_KEYEDHASH
250   #define ALG_KEYEDHASH               ALG_YES
251   #endif
252   #ifndef ALG_MGF1
253   #define ALG_MGF1                    ALG_YES
254   #endif
255   #ifndef ALG_OAEP
256   #define ALG_OAEP                    (ALG_YES && ALG_RSA)
257   #endif
258   #ifndef ALG_OFB
259   #define ALG_OFB                     ALG_YES
260   #endif
261   #ifndef ALG_RSA
262   #define ALG_RSA                     ALG_YES
263   #endif
264   #ifndef ALG_RSAES
265   #define ALG_RSAES                   (ALG_YES && ALG_RSA)
266   #endif
267   #ifndef ALG_RSAPSS
268   #define ALG_RSAPSS                  (ALG_YES && ALG_RSA)
269   #endif
270   #ifndef ALG_RSASSA
271   #define ALG_RSASSA                  (ALG_YES && ALG_RSA)
272   #endif
273   #ifndef ALG_SHA
274   #define ALG_SHA                     ALG_NO      /* Not specified by vendor */
275   #endif
276   #ifndef ALG_SHA1
277   #define ALG_SHA1                    ALG_YES
278   #endif
279   #ifndef ALG_SHA256
280   #define ALG_SHA256                  ALG_YES
281   #endif
282   #ifndef ALG_SHA384
```

```
283   #define ALG_SHA384                      ALG_YES
284   #endif
285   #ifndef ALG_SHA3_256
286   #define ALG_SHA3_256                    ALG_NO      /* Not specified by vendor */
287   #endif
288   #ifndef ALG_SHA3_384
289   #define ALG_SHA3_384                    ALG_NO      /* Not specified by vendor */
290   #endif
291   #ifndef ALG_SHA3_512
292   #define ALG_SHA3_512                    ALG_NO      /* Not specified by vendor */
293   #endif
294   #ifndef ALG_SHA512
295   #define ALG_SHA512                      ALG_NO
296   #endif
297   #ifndef ALG_SM2
298   #define ALG_SM2                         (ALG_NO && ALG_ECC)
299   #endif
300   #ifndef ALG_SM3_256
301   #define ALG_SM3_256                     ALG_NO
302   #endif
303   #ifndef ALG_SM4
304   #define ALG_SM4                         ALG_YES
305   #endif
306   #ifndef ALG_SYMCIPHER
307   #define ALG_SYMCIPHER                   ALG_YES
308   #endif
309   #ifndef ALG_TDES
310   #define ALG_TDES                        ALG_NO
311   #endif
312   #ifndef ALG_XOR
313   #define ALG_XOR                         ALG_YES
314   #endif
```

Table 1:3 - Defines for RSA Asymmetric Cipher Algorithm Constants

```
315   #ifndef RSA_1024
316   #define RSA_1024                    (ALG_RSA && YES)
317   #endif
318   #ifndef RSA_2048
319   #define RSA_2048                    (ALG_RSA && YES)
320   #endif
321   #ifndef RSA_3072
322   #define RSA_3072                    (ALG_RSA && NO)
323   #endif
324   #ifndef RSA_4096
325   #define RSA_4096                    (ALG_RSA && NO)
326   #endif
```

Table 1:21 - Defines for AES Symmetric Cipher Algorithm Constants

```
327   #ifndef AES_128
328   #define AES_128                     (ALG_AES && YES)
329   #endif
330   #ifndef AES_192
331   #define AES_192                     (ALG_AES && NO)
332   #endif
333   #ifndef AES_256
334   #define AES_256                     (ALG_AES && YES)
335   #endif
```

Table 1:22 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
336   #ifndef SM4_128
337   #define SM4_128                     (ALG_SM4 && YES)
```

```
#endif
```

Table 1:23 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
#ifndef CAMELLIA_128
#define CAMELLIA_128                    (ALG_CAMELLIA && YES)
#endif
#ifndef CAMELLIA_192
#define CAMELLIA_192                    (ALG_CAMELLIA && NO)
#endif
#ifndef CAMELLIA_256
#define CAMELLIA_256                    (ALG_CAMELLIA && YES)
#endif
```

Table 1:24 - Defines for TDES Symmetric Cipher Algorithm Constants

```
#ifndef TDES_128
#define TDES_128                    (ALG_TDES && YES)
#endif
#ifndef TDES_192
#define TDES_192                    (ALG_TDES && YES)
#endif
```

Table 0:5 - Defines for Implemented Commands

```
#ifndef CC_ACT_SetTimeout
#define CC_ACT_SetTimeout                   CC_YES
#endif
#ifndef CC_AC_GetCapability
#define CC_AC_GetCapability                 CC_YES
#endif
#ifndef CC_AC_Send
#define CC_AC_Send                          CC_YES
#endif
#ifndef CC_ActivateCredential
#define CC_ActivateCredential               CC_YES
#endif
#ifndef CC_Certify
#define CC_Certify                          CC_YES
#endif
#ifndef CC_CertifyCreation
#define CC_CertifyCreation                  CC_YES
#endif
#ifndef CC_CertifyX509
#define CC_CertifyX509                      CC_YES
#endif
#ifndef CC_ChangeEPS
#define CC_ChangeEPS                        CC_YES
#endif
#ifndef CC_ChangePPS
#define CC_ChangePPS                        CC_YES
#endif
#ifndef CC_Clear
#define CC_Clear                            CC_YES
#endif
#ifndef CC_ClearControl
#define CC_ClearControl                     CC_YES
#endif
#ifndef CC_ClockRateAdjust
#define CC_ClockRateAdjust                  CC_YES
#endif
#ifndef CC_ClockSet
#define CC_ClockSet                         CC_YES
#endif
```

```
393  #ifndef CC_Commit
394  #define CC_Commit                          (CC_YES && ALG_ECC)
395  #endif
396  #ifndef CC_ContextLoad
397  #define CC_ContextLoad                     CC_YES
398  #endif
399  #ifndef CC_ContextSave
400  #define CC_ContextSave                     CC_YES
401  #endif
402  #ifndef CC_Create
403  #define CC_Create                          CC_YES
404  #endif
405  #ifndef CC_CreateLoaded
406  #define CC_CreateLoaded                    CC_YES
407  #endif
408  #ifndef CC_CreatePrimary
409  #define CC_CreatePrimary                   CC_YES
410  #endif
411  #ifndef CC_DictionaryAttackLockReset
412  #define CC_DictionaryAttackLockReset       CC_YES
413  #endif
414  #ifndef CC_DictionaryAttackParameters
415  #define CC_DictionaryAttackParameters      CC_YES
416  #endif
417  #ifndef CC_Duplicate
418  #define CC_Duplicate                       CC_YES
419  #endif
420  #ifndef CC_ECC_Parameters
421  #define CC_ECC_Parameters                  (CC_YES && ALG_ECC)
422  #endif
423  #ifndef CC_ECDH_KeyGen
424  #define CC_ECDH_KeyGen                     (CC_YES && ALG_ECC)
425  #endif
426  #ifndef CC_ECDH_ZGen
427  #define CC_ECDH_ZGen                       (CC_YES && ALG_ECC)
428  #endif
429  #ifndef CC_EC_Ephemeral
430  #define CC_EC_Ephemeral                    (CC_YES && ALG_ECC)
431  #endif
432  #ifndef CC_EncryptDecrypt
433  #define CC_EncryptDecrypt                  CC_YES
434  #endif
435  #ifndef CC_EncryptDecrypt2
436  #define CC_EncryptDecrypt2                 CC_YES
437  #endif
438  #ifndef CC_EventSequenceComplete
439  #define CC_EventSequenceComplete           CC_YES
440  #endif
441  #ifndef CC_EvictControl
442  #define CC_EvictControl                    CC_YES
443  #endif
444  #ifndef CC_FieldUpgradeData
445  #define CC_FieldUpgradeData                CC_NO
446  #endif
447  #ifndef CC_FieldUpgradeStart
448  #define CC_FieldUpgradeStart               CC_NO
449  #endif
450  #ifndef CC_FirmwareRead
451  #define CC_FirmwareRead                    CC_NO
452  #endif
453  #ifndef CC_FlushContext
454  #define CC_FlushContext                    CC_YES
455  #endif
456  #ifndef CC_GetCapability
457  #define CC_GetCapability                   CC_YES
458  #endif
```

```
459  #ifndef CC_GetCommandAuditDigest
460  #define CC_GetCommandAuditDigest            CC_YES
461  #endif
462  #ifndef CC_GetRandom
463  #define CC_GetRandom                        CC_YES
464  #endif
465  #ifndef CC_GetSessionAuditDigest
466  #define CC_GetSessionAuditDigest            CC_YES
467  #endif
468  #ifndef CC_GetTestResult
469  #define CC_GetTestResult                    CC_YES
470  #endif
471  #ifndef CC_GetTime
472  #define CC_GetTime                          CC_YES
473  #endif
474  #ifndef CC_HMAC
475  #define CC_HMAC                             (CC_YES && !ALG_CMAC)
476  #endif
477  #ifndef CC_HMAC_Start
478  #define CC_HMAC_Start                       (CC_YES && !ALG_CMAC)
479  #endif
480  #ifndef CC_Hash
481  #define CC_Hash                             CC_YES
482  #endif
483  #ifndef CC_HashSequenceStart
484  #define CC_HashSequenceStart                CC_YES
485  #endif
486  #ifndef CC_HierarchyChangeAuth
487  #define CC_HierarchyChangeAuth              CC_YES
488  #endif
489  #ifndef CC_HierarchyControl
490  #define CC_HierarchyControl                 CC_YES
491  #endif
492  #ifndef CC_Import
493  #define CC_Import                           CC_YES
494  #endif
495  #ifndef CC_IncrementalSelfTest
496  #define CC_IncrementalSelfTest              CC_YES
497  #endif
498  #ifndef CC_Load
499  #define CC_Load                             CC_YES
500  #endif
501  #ifndef CC_LoadExternal
502  #define CC_LoadExternal                     CC_YES
503  #endif
504  #ifndef CC_MAC
505  #define CC_MAC                              (CC_YES && ALG_CMAC)
506  #endif
507  #ifndef CC_MAC_Start
508  #define CC_MAC_Start                        (CC_YES && ALG_CMAC)
509  #endif
510  #ifndef CC_MakeCredential
511  #define CC_MakeCredential                   CC_YES
512  #endif
513  #ifndef CC_NV_Certify
514  #define CC_NV_Certify                       CC_YES
515  #endif
516  #ifndef CC_NV_ChangeAuth
517  #define CC_NV_ChangeAuth                    CC_YES
518  #endif
519  #ifndef CC_NV_DefineSpace
520  #define CC_NV_DefineSpace                   CC_YES
521  #endif
522  #ifndef CC_NV_Extend
523  #define CC_NV_Extend                        CC_YES
524  #endif
```

```
525   #ifndef CC_NV_GlobalWriteLock
526   #define CC_NV_GlobalWriteLock               CC_YES
527   #endif
528   #ifndef CC_NV_Increment
529   #define CC_NV_Increment                     CC_YES
530   #endif
531   #ifndef CC_NV_Read
532   #define CC_NV_Read                          CC_YES
533   #endif
534   #ifndef CC_NV_ReadLock
535   #define CC_NV_ReadLock                      CC_YES
536   #endif
537   #ifndef CC_NV_ReadPublic
538   #define CC_NV_ReadPublic                    CC_YES
539   #endif
540   #ifndef CC_NV_SetBits
541   #define CC_NV_SetBits                       CC_YES
542   #endif
543   #ifndef CC_NV_UndefineSpace
544   #define CC_NV_UndefineSpace                 CC_YES
545   #endif
546   #ifndef CC_NV_UndefineSpaceSpecial
547   #define CC_NV_UndefineSpaceSpecial          CC_YES
548   #endif
549   #ifndef CC_NV_Write
550   #define CC_NV_Write                         CC_YES
551   #endif
552   #ifndef CC_NV_WriteLock
553   #define CC_NV_WriteLock                     CC_YES
554   #endif
555   #ifndef CC_ObjectChangeAuth
556   #define CC_ObjectChangeAuth                 CC_YES
557   #endif
558   #ifndef CC_PCR_Allocate
559   #define CC_PCR_Allocate                     CC_YES
560   #endif
561   #ifndef CC_PCR_Event
562   #define CC_PCR_Event                        CC_YES
563   #endif
564   #ifndef CC_PCR_Extend
565   #define CC_PCR_Extend                       CC_YES
566   #endif
567   #ifndef CC_PCR_Read
568   #define CC_PCR_Read                         CC_YES
569   #endif
570   #ifndef CC_PCR_Reset
571   #define CC_PCR_Reset                        CC_YES
572   #endif
573   #ifndef CC_PCR_SetAuthPolicy
574   #define CC_PCR_SetAuthPolicy                CC_YES
575   #endif
576   #ifndef CC_PCR_SetAuthValue
577   #define CC_PCR_SetAuthValue                 CC_YES
578   #endif
579   #ifndef CC_PP_Commands
580   #define CC_PP_Commands                      CC_YES
581   #endif
582   #ifndef CC_PolicyAuthValue
583   #define CC_PolicyAuthValue                  CC_YES
584   #endif
585   #ifndef CC_PolicyAuthorize
586   #define CC_PolicyAuthorize                  CC_YES
587   #endif
588   #ifndef CC_PolicyAuthorizeNV
589   #define CC_PolicyAuthorizeNV                CC_YES
590   #endif
```

```
591  #ifndef CC_PolicyCommandCode
592  #define CC_PolicyCommandCode                CC_YES
593  #endif
594  #ifndef CC_PolicyCounterTimer
595  #define CC_PolicyCounterTimer               CC_YES
596  #endif
597  #ifndef CC_PolicyCpHash
598  #define CC_PolicyCpHash                     CC_YES
599  #endif
600  #ifndef CC_PolicyDuplicationSelect
601  #define CC_PolicyDuplicationSelect          CC_YES
602  #endif
603  #ifndef CC_PolicyGetDigest
604  #define CC_PolicyGetDigest                  CC_YES
605  #endif
606  #ifndef CC_PolicyLocality
607  #define CC_PolicyLocality                   CC_YES
608  #endif
609  #ifndef CC_PolicyNV
610  #define CC_PolicyNV                         CC_YES
611  #endif
612  #ifndef CC_PolicyNameHash
613  #define CC_PolicyNameHash                   CC_YES
614  #endif
615  #ifndef CC_PolicyNvWritten
616  #define CC_PolicyNvWritten                  CC_YES
617  #endif
618  #ifndef CC_PolicyOR
619  #define CC_PolicyOR                         CC_YES
620  #endif
621  #ifndef CC_PolicyPCR
622  #define CC_PolicyPCR                        CC_YES
623  #endif
624  #ifndef CC_PolicyPassword
625  #define CC_PolicyPassword                   CC_YES
626  #endif
627  #ifndef CC_PolicyPhysicalPresence
628  #define CC_PolicyPhysicalPresence           CC_YES
629  #endif
630  #ifndef CC_PolicyRestart
631  #define CC_PolicyRestart                    CC_YES
632  #endif
633  #ifndef CC_PolicySecret
634  #define CC_PolicySecret                     CC_YES
635  #endif
636  #ifndef CC_PolicySigned
637  #define CC_PolicySigned                     CC_YES
638  #endif
639  #ifndef CC_PolicyTemplate
640  #define CC_PolicyTemplate                   CC_YES
641  #endif
642  #ifndef CC_PolicyTicket
643  #define CC_PolicyTicket                     CC_YES
644  #endif
645  #ifndef CC_Policy_AC_SendSelect
646  #define CC_Policy_AC_SendSelect             CC_YES
647  #endif
648  #ifndef CC_Quote
649  #define CC_Quote                            CC_YES
650  #endif
651  #ifndef CC_RSA_Decrypt
652  #define CC_RSA_Decrypt                      (CC_YES && ALG_RSA)
653  #endif
654  #ifndef CC_RSA_Encrypt
655  #define CC_RSA_Encrypt                      (CC_YES && ALG_RSA)
656  #endif
```

```
657  #ifndef CC_ReadClock
658  #define CC_ReadClock                        CC_YES
659  #endif
660  #ifndef CC_ReadPublic
661  #define CC_ReadPublic                       CC_YES
662  #endif
663  #ifndef CC_Rewrap
664  #define CC_Rewrap                           CC_YES
665  #endif
666  #ifndef CC_SelfTest
667  #define CC_SelfTest                         CC_YES
668  #endif
669  #ifndef CC_SequenceComplete
670  #define CC_SequenceComplete                 CC_YES
671  #endif
672  #ifndef CC_SequenceUpdate
673  #define CC_SequenceUpdate                   CC_YES
674  #endif
675  #ifndef CC_SetAlgorithmSet
676  #define CC_SetAlgorithmSet                  CC_YES
677  #endif
678  #ifndef CC_SetCommandCodeAuditStatus
679  #define CC_SetCommandCodeAuditStatus        CC_YES
680  #endif
681  #ifndef CC_SetPrimaryPolicy
682  #define CC_SetPrimaryPolicy                 CC_YES
683  #endif
684  #ifndef CC_Shutdown
685  #define CC_Shutdown                         CC_YES
686  #endif
687  #ifndef CC_Sign
688  #define CC_Sign                             CC_YES
689  #endif
690  #ifndef CC_StartAuthSession
691  #define CC_StartAuthSession                 CC_YES
692  #endif
693  #ifndef CC_Startup
694  #define CC_Startup                          CC_YES
695  #endif
696  #ifndef CC_StirRandom
697  #define CC_StirRandom                       CC_YES
698  #endif
699  #ifndef CC_TestParms
700  #define CC_TestParms                        CC_YES
701  #endif
702  #ifndef CC_Unseal
703  #define CC_Unseal                           CC_YES
704  #endif
705  #ifndef CC_Vendor_TCG_Test
706  #define CC_Vendor_TCG_Test                  CC_YES
707  #endif
708  #ifndef CC_VerifySignature
709  #define CC_VerifySignature                  CC_YES
710  #endif
711  #ifndef CC_ZGen_2Phase
712  #define CC_ZGen_2Phase                      (CC_YES && ALG_ECC)
713  #endif
714  #endif // _TPM_PROFILE_H_
```

A.3 TpmSizeChecks.c

A.3.1. Includes, Defines, and Types

```
1    #include    "Tpm.h"
```

```
#include    <stdio.h>
#include    <assert.h>
#if RUNTIME_SIZE_CHECKS
#if TABLE_DRIVEN_MARSHAL
extern uint32_t    MarshalDataSize;
#endif

static      int once = 0;

//** TpmSizeChecks()
// This function is used during the development process to make sure that the
// vendor-specific values result in a consistent implementation. When possible,
// the code contains #if to do compile-time checks. However, in some cases, the
// values require the use of "sizeof()" and that can't be used in an #if.
BOOL
TpmSizeChecks(
    void
    )
{
    BOOL        PASS = TRUE;
#if DEBUG
//
    if(once++ != 0)
        return 1;
    {
        UINT32      maxAsymSecurityStrength = MAX_ASYM_SECURITY_STRENGTH;
        UINT32      maxHashSecurityStrength = MAX_HASH_SECURITY_STRENGTH;
        UINT32      maxSymSecurityStrength = MAX_SYM_SECURITY_STRENGTH;
        UINT32      maxSecurityStrengthBits = MAX_SECURITY_STRENGTH_BITS;
        UINT32      proofSize = PROOF_SIZE;
        UINT32      compliantProofSize = COMPLIANT_PROOF_SIZE;
        UINT32      compliantPrimarySeedSize = COMPLIANT_PRIMARY_SEED_SIZE;
        UINT32      primarySeedSize = PRIMARY_SEED_SIZE;

        UINT32      cmacState = sizeof(tpmCmacState_t);
        UINT32      hashState = sizeof(HASH_STATE);
        UINT32      keyScheduleSize = sizeof(tpmCryptKeySchedule_t);
    //
        NOT_REFERENCED(cmacState);
        NOT_REFERENCED(hashState);
        NOT_REFERENCED(keyScheduleSize);
        NOT_REFERENCED(maxAsymSecurityStrength);
        NOT_REFERENCED(maxHashSecurityStrength);
        NOT_REFERENCED(maxSymSecurityStrength);
        NOT_REFERENCED(maxSecurityStrengthBits);
        NOT_REFERENCED(proofSize);
        NOT_REFERENCED(compliantProofSize);
        NOT_REFERENCED(compliantPrimarySeedSize);
        NOT_REFERENCED(primarySeedSize);

        {
            TPMT_SENSITIVE          *p;
            // This assignment keeps compiler from complaining about a conditional
            // comparison being between two constants
            UINT16                   max_rsa_key_bytes = MAX_RSA_KEY_BYTES;
            if((max_rsa_key_bytes / 2) != (sizeof(p->sensitive.rsa.t.buffer) / 5))
            {
                printf("Sensitive part of TPMT_SENSITIVE is undersized. May be caused"
                       " by use of wrong version of Part 2.\n");
                PASS = FALSE;
            }
        }
#if TABLE_DRIVEN_MARSHAL
        printf("sizeof(MarshalData) = %zu\n", sizeof(MarshalData_st));
#endif

```

```
68          printf("Size of OBJECT = %zu\n", sizeof(OBJECT));
69          printf("Size of components in TPMT_SENSITIVE = %zu\n",
    sizeof(TPMT_SENSITIVE));
70          printf("    TPMI_ALG_PUBLIC                 %zu\n", sizeof(TPMI_ALG_PUBLIC));
71          printf("    TPM2B_AUTH                      %zu\n", sizeof(TPM2B_AUTH));
72          printf("    TPM2B_DIGEST                    %zu\n", sizeof(TPM2B_DIGEST));
73          printf("    TPMU_SENSITIVE_COMPOSITE        %zu\n",
74                 sizeof(TPMU_SENSITIVE_COMPOSITE));
75      }
76          // Make sure that the size of the context blob is large enough for the largest
77          // context
78          // TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
79          // implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
80          // determine the amount of data in the encrypted data. That part is not
81          // independently sized. This makes the actual size 2 bytes smaller than
82          // calculated using Part 2. Since this is opaque to the caller, it is not
83          // necessary to fix. The actual size is returned by TPM2_GetCapabilties().
84
85          // Initialize output handle.  At the end of command action, the output
86          // handle of an object will be replaced, while the output handle
87          // for a session will be the same as input
88
89          // Get the size of fingerprint in context blob.  The sequence value in
90          // TPMS_CONTEXT structure is used as the fingerprint
91      {
92          UINT32  fingerprintSize = sizeof(UINT64);
93          UINT32  integritySize = sizeof(UINT16)
94              + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
95          UINT32  biggestObject = MAX(MAX(sizeof(HASH_OBJECT), sizeof(OBJECT)),
96                                      sizeof(SESSION));
97          UINT32  biggestContext = fingerprintSize + integritySize + biggestObject;
98
99          // round required size up to nearest 8 byte boundary.
100         biggestContext = 8 * ((biggestContext + 7) / 8);
101
102         if(MAX_CONTEXT_SIZE != biggestContext)
103         {
104             printf("MAX_CONTEXT_SIZE should be changed to %d (%d)\n",
105                    biggestContext, MAX_CONTEXT_SIZE);
106             PASS = FALSE;
107         }
108     }
109     {
110         union u
111         {
112             TPMA_OBJECT             attributes;
113             UINT32                  uint32Value;
114         } u;
115         // these are defined so that compiler doesn't complain about conditional
116         // expressions comparing two constants.
117         int                         aSize = sizeof(u.attributes);
118         int                         uSize = sizeof(u.uint32Value);
119         u.uint32Value = 0;
120         SET_ATTRIBUTE(u.attributes, TPMA_OBJECT, Reserved_bit_at_0);
121         if(u.uint32Value != 1)
122         {
123             printf("The bit allocation in a TPMA_OBJECT is not as expected");
124             PASS = FALSE;
125         }
126         if(aSize != uSize)  // comparison of two sizeof() values annoys compiler
127         {
128             printf("A TPMA_OBJECT is not the expected size.");
129             PASS = FALSE;
130         }
131     }
132     // Check that the platform implements each of the ACT that the TPM thinks
```

```
133         {
134             uint32_t            act;
135             for(act = 0; act < 16; act++)
136             {
137                 switch(act)
138                 {
139                     FOR_EACH_ACT(CASE_ACT_NUMBER)
140                         if(!_plat__ACT_GetImplemented(act))
141                         {
142                             printf("TPM_RH_ACT_%1X is not implemented by platform\n",
143                                    act);
144                             PASS = FALSE;
145                         }
146                     default:
147                         break;
148                 }
149             }
150         }
151     #endif // DEBUG
152         return (PASS);
153     }
154     #endif // RUNTIME_SIZE_CHECKS
```

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 HASH_LIB_DEFINED
#define HASH_LIB_DEFINED
#define HASH_LIB_OSSL
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <openssl/sm3.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
#define tpmHashStateSM3_256_t     SM3_CTX
```

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.

```
#define HASH_ALIGNMENT  RADIX_BYTES
```

Initialize the hash context

```
17  #define HASH_START_METHOD_DEF   void (HASH_START_METHOD)(PANY_HASH_STATE state)
18  #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.

```
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_SHA_512  memcpy
81   #define tpmHashStart_SM3_256        sm3_init
82   #define tpmHashData_SM3_256         sm3_update
83   #define tpmHashEnd_SM3_256          sm3_final
84   #define tpmHashStateCopy_SM3_256    memcpy
85   #define tpmHashStateExport_SM3_256  memcpy
86   #define tpmHashStateImport_SM3_256  memcpy
87   #endif // _CRYPT_HASH_C_
88   #define LibHashInit()
```

This definition would change if there were something to report

```
89   #define HashLibSimulationEnd()
90   #endif // HASH_LIB_DEFINED
```

B.2.2.2. TpmToOsslMath.h

B.2.2.2.1. Introduction

This file contains the structure definitions used for ECC in the LibTomCrypt() version of the code. These definitions would change, based on the library. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
#ifndef MATH_LIB_DEFINED
#define MATH_LIB_DEFINED
#define MATH_LIB_OSSL
#include <openssl/evp.h>
#include <openssl/ec.h>
#if OPENSSL_VERSION_NUMBER >= 0x10200000L
    // Check the bignum_st definition in crypto/bn/bn_lcl.h and either update the
    // version check or provide the new definition for this version.
#   error Untested OpenSSL version
#elif OPENSSL_VERSION_NUMBER >= 0x10100000L
    // from crypto/bn/bn_lcl.h
    struct bignum_st {
        BN_ULONG *d;                /* Pointer to an array of 'BN_BITS2' bit
                                    * chunks. */
        int top;                    /* Index of last used d +1. */
                                    /* The next are internal book keeping for
bn_expand. */
        int dmax;                   /* Size of the d array. */
        int neg;                    /* one if the number is negative */
        int flags;
    };
#endif // OPENSSL_VERSION_NUMBER
#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

```
#if    defined THIRTY_TWO_BIT && (RADIX_BITS != 32)  \
    || ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \
        && (RADIX_BITS != 64))
#   error Ossl library is using different radix
#endif
```

Allocate a local BIGNUM value. For the allocation, a *bigNum* structure is created as is a local BIGNUM. The *bigNum* is initialized and then the BIGNUM is set to reference the local value.

```
#define BIG_VAR(name, bits)                                          \
    BN_VAR(name##Bn, (bits));                                        \
    BIGNUM          _##name;                                         \
    BIGNUM          *name = BigInitialized(&_##name,                 \
                                BnInit(name##Bn,                     \
                                BYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d))))
```

Allocate a BIGNUM and initialize with the values in a *bigNum* initializer

```
#define BIG_INITIALIZED(name, initializer)                      \
    BIGNUM          _##name;                                    \
    BIGNUM          *name = BigInitialized(&_##name, initializer)
typedef struct
{
    const ECC_CURVE_DATA    *C;     // the TPM curve values
    EC_GROUP                *G;     // group parameters
```

```
41      BN_CTX                  *CTX;   // the context for the math (this might not be
42                                      // the context in which the curve was created>;
43  } OSSL_CURVE_DATA;
44  typedef OSSL_CURVE_DATA      *bigCurve;
45  #define AccessCurveData(E)       ((E)->C)
46  #include "TpmToOsslSupport_fp.h"
```

Start and end a context within which the OpenSSL memory management works

```
47  #define OSSL_ENTER()    BN_CTX          *CTX = OsslContextEnter()
48  #define OSSL_LEAVE()    OsslContextLeave(CTX)
```

Start and end a context that spans multiple ECC functions. This is used so that the group for the curve can persist across multiple frames.

```
49  #define CURVE_INITIALIZED(name, initializer)                        \
50      OSSL_CURVE_DATA     _##name;                                    \
51      bigCurve            name =  BnCurveInitialize(&_##name, initializer)
52  #define CURVE_FREE(name)                BnCurveFree(name)
```

Start and end a local stack frame within the context of the curve frame

```
53  #define ECC_ENTER()     BN_CTX          *CTX = OsslPushContext(E->CTX)
54  #define ECC_LEAVE()     OsslPopContext(CTX)
55  #define BN_NEW()        BnNewVariable(CTX)
```

This definition would change if there were something to report

```
56  #define MathLibSimulationEnd()
57  #endif // MATH_LIB_DEFINED
```

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. All of the library-dependent headers should have the same guard to that only the first one gets defined.

```
#ifndef SYM_LIB_DEFINED
#define SYM_LIB_DEFINED
#define SYM_LIB_OSSL
#include <openssl/aes.h>
#include <openssl/des.h>
#include <openssl/sm4.h>
#include <openssl/camellia.h>
#include <openssl/bn.h>
#include <openssl/ossl_typ.h>
```

B.2.2.3.2. Links to the OpenSSL symmetric algorithms

The Crypt functions that call the block encryption function use the parameters in the order:

a) *keySchedule*

b) in buffer

c) out buffer Since open SSL uses the order in *encryptoCall_t* above, need to swizzle the values to the order required by the library.

```
#define SWIZZLE(keySchedule, in, out)                                   \
    (const BYTE *)(in), (BYTE *)(out), (void *)(keySchedule)
```

Define the order of parameters to the library functions that do block encryption and decryption.

```
typedef void(*TpmCryptSetSymKeyCall_t)(
    const BYTE  *in,
    BYTE        *out,
    void *keySchedule
    );
#define SYM_ALIGNMENT   RADIX_BYTES
```

B.2.2.3.3. Links to the OpenSSL AES code

Macros to set up the encryption/decryption key schedules

AES:

```
#define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)          \
    AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
#define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule)          \
    AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly. Currently, only used by CryptSym.c and CryptRand.c

When using these calls, to call the AES block encryption code, the caller should use: TpmCryptEncryptAES(SWIZZLE(*keySchedule*, in, out));

```
22  #define TpmCryptEncryptAES          AES_encrypt
23  #define TpmCryptDecryptAES          AES_decrypt
24  #define tpmKeyScheduleAES           AES_KEY
```

B.2.2.3.4. Links to the OpenSSL DES code

```
25  #if ALG_TDES
26  #include "TpmToOsslDesSupport_fp.h"
27  #endif
28  #define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule)                    \
29      TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
30  #define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule)                    \
31      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

```
32  #define TpmCryptEncryptTDES         TDES_encrypt
33  #define TpmCryptDecryptTDES         TDES_decrypt
34  #define tpmKeyScheduleTDES          DES_key_schedule
```

B.2.2.3.5. Links to the OpenSSL SM4 code

Macros to set up the encryption/decryption key schedules

```
35  #define TpmCryptSetEncryptKeySM4(key, keySizeInBits, schedule)                     \
36      SM4_set_key((key), (tpmKeyScheduleSM4 *)(schedule))
37  #define TpmCryptSetDecryptKeySM4(key, keySizeInBits, schedule)                     \
38      SM4_set_key((key), (tpmKeyScheduleSM4 *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly.

```
39  #define TpmCryptEncryptSM4          SM4_encrypt
40  #define TpmCryptDecryptSM4          SM4_decrypt
41  #define tpmKeyScheduleSM4           SM4_KEY
```

B.2.2.3.6. Links to the OpenSSL CAMELLIA code

Macros to set up the encryption/decryption key schedules

```
42  #define TpmCryptSetEncryptKeyCAMELLIA(key, keySizeInBits, schedule)                \
43      Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA *)(schedule))
44  #define TpmCryptSetDecryptKeyCAMELLIA(key, keySizeInBits, schedule)                \
45      Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly.

```
46  #define TpmCryptEncryptCAMELLIA          Camellia_encrypt
47  #define TpmCryptDecryptCAMELLIA          Camellia_decrypt
48  #define tpmKeyScheduleCAMELLIA           CAMELLIA_KEY
```

Forward reference

```
49  typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;
```

This definition would change if there were something to report

```
50  #define SymLibSimulationEnd()
51  #endif // SYM_LIB_DEFINED
```

B.2.3. Source Files

B.2.3.1. TpmToOsslDesSupport.c

B.2.3.1.1. Introduction

The functions in this file are used for initialization of the interface to the OpenSSL library.

B.2.3.1.2. Defines and Includes

```
#include "Tpm.h"
#if (defined SYM_LIB_OSSL) && ALG_TDES
```

B.2.3.1.3. Functions

B.2.3.1.3.1. TDES_set_encyrpt_key()

This function makes creation of a TDES key look like the creation of a key for any of the other OpenSSL block ciphers. It will create three key schedules, one for each of the DES keys. If there are only two keys, then the third schedule is a copy of the first.

```
void
TDES_set_encrypt_key(
    const BYTE                  *key,
    UINT16                       keySizeInBits,
    tpmKeyScheduleTDES          *keySchedule
    )
{
    DES_set_key_unchecked((const_DES_cblock *)key, &keySchedule[0]);
    DES_set_key_unchecked((const_DES_cblock *)&key[8], &keySchedule[1]);
    // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
    // the schedule for K3
    if(keySizeInBits == 128)
        keySchedule[2] = keySchedule[0];
    else
        DES_set_key_unchecked((const_DES_cblock *)&key[16],
                              &keySchedule[2]);
}
```

B.2.3.1.3.2. TDES_encyrpt()

The TPM code uses one key schedule. For TDES, the schedule contains three schedules. OpenSSL wants the schedules referenced separately. This function does that.

```
void TDES_encrypt(
    const BYTE              *in,
    BYTE                    *out,
    tpmKeyScheduleTDES      *ks
    )
{
    DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
                     &ks[0], &ks[1], &ks[2],
                     DES_ENCRYPT);
}
```

B.2.3.1.3.3. TDES_decrypt()

As with TDES_encypt() this function bridges between the TPM single schedule model and the OpenSSL three schedule model.

```
30  void TDES_decrypt(
31      const BYTE          *in,
32      BYTE                *out,
33      tpmKeyScheduleTDES   *ks
34      )
35  {
36      DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
37                       &ks[0], &ks[1], &ks[2],
38                       DES_DECRYPT);
39  }
40  #endif // SYM_LIB_OSSL
```


B.2.3.2. TpmToOsslMath.c

B.2.3.2.1. Introduction

The functions in this file provide the low-level interface between the TPM code and the big number and elliptic curve math routines in OpenSSL.

Most math on big numbers require a context. The context contains the memory in which OpenSSL creates and manages the big number values. When a OpenSSL math function will be called that modifies a BIGNUM value, that value must be created in an OpenSSL context. The first line of code in such a function must be: OSSL_ENTER(); and the last operation before returning must be OSSL_LEAVE(). OpenSSL variables can then be created with BnNewVariable(). Constant values to be used by OpenSSL are created from the *bigNum* values passed to the functions in this file. Space for the BIGNUM control block is allocated in the stack of the function and then it is initialized by calling BigInitialized(). That function sets up the values in the BIGNUM structure and sets the data pointer to point to the data in the bignum_t. This is only used when the value is known to be a constant in the called function.

Because the allocations of constants is on the local stack and the OSSL_ENTER()/OSSL_LEAVE() pair flushes everything created in OpenSSL memory, there should be no chance of a memory leak.

B.2.3.2.2. Includes and Defines

```
1   #include "Tpm.h"
2   #ifdef 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 uses a different control structure but the same data layout -- an array of native-endian words in little-endian order.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure because value will not fit or OpenSSL variable doesn't exist

```
4   BOOL
5   OsslToTpmBn(
6       bigNum          bn,
7       BIGNUM          *osslBn
8       )
9   {
10      VERIFY(osslBn != NULL);
11      // If the bn is NULL, it means that an output value pointer was NULL meaning that
12      // the results is simply to be discarded.
13      if(bn != NULL)
14      {
15          int         i;
16      //
17          VERIFY((unsigned)osslBn->top <= BnGetAllocated(bn));
18          for(i = 0; i < osslBn->top; i++)
19              bn->d[i] = osslBn->d[i];
20          BnSetTop(bn, osslBn->top);
21      }
22      return TRUE;
```

```
23  Error:
24      return FALSE;
25  }
```

B.2.3.2.3.2. BigInitialized()

This function initializes an OSSL BIGNUM from a TPM *bigConst*. Do not use this for values that are passed to OpenSLL when they are not declared as const in the function prototype. Instead, use BnNewVariable().

```
26  BIGNUM *
27  BigInitialized(
28      BIGNUM              *toInit,
29      bigConst             initializer
30      )
31  {
32      if(initializer == NULL)
33          FAIL(FATAL_ERROR_PARAMETER);
34      if(toInit == NULL || initializer == NULL)
35          return NULL;
36      toInit->d = (BN_ULONG *)&initializer->d[0];
37      toInit->dmax = (int)initializer->allocated;
38      toInit->top = (int)initializer->size;
39      toInit->neg = 0;
40      toInit->flags = 0;
41      return toInit;
42  }
43  #ifndef OSSL_DEBUG
44  #   define BIGNUM_PRINT(label, bn, eol)
45  #   define DEBUG_PRINT(x)
46  #else
47  #   define DEBUG_PRINT(x)   printf("%s", x)
48  #   define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))
```

B.2.3.2.3.3. BIGNUM_print()

```
49  static void
50  BIGNUM_print(
51      const char      *label,
52      const BIGNUM    *a,
53      BOOL             eol
54      )
55  {
56      BN_ULONG        *d;
57      int              i;
58      int              notZero = FALSE;
59
60      if(label != NULL)
61          printf("%s", label);
62      if(a == NULL)
63      {
64          printf("NULL");
65          goto done;
66      }
67      if (a->neg)
68          printf("-");
69      for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
70      {
71          int         j;
72          BN_ULONG    l = *d--;
73          for(j = BN_BITS2 - 8; j >= 0; j -= 8)
74          {
75              BYTE    b = (BYTE)((l >> j) & 0xFF);
```

```
76              notZero = notZero || (b != 0);
77              if(notZero)
78                  printf("%02x", b);
79          }
80          if(!notZero)
81              printf("0");
82      }
83  done:
84      if(eol)
85          printf("\n");
86      return;
87  }
88  #endif
```

B.2.3.2.3.4. BnNewVariable()

This function allocates a new variable in the provided context. If the context does not exist or the allocation fails, it is a catastrophic failure.

```
89  static BIGNUM *
90  BnNewVariable(
91      BN_CTX          *CTX
92  )
93  {
94      BIGNUM          *new;
95  //
96      // This check is intended to protect against calling this function without
97      // having initialized the CTX.
98      if((CTX == NULL) || ((new = BN_CTX_get(CTX)) == NULL))
99          FAIL(FATAL_ERROR_ALLOCATION);
100     return new;
101 }
102 #if LIBRARY_COMPATIBILITY_CHECK
```

B.2.3.2.3.5. MathLibraryCompatibilityCheck()

```
103 BOOL
104 MathLibraryCompatibilityCheck(
105     void
106     )
107 {
108     OSSL_ENTER();
109     BIGNUM              *osslTemp = BnNewVariable(CTX);
110     crypt_uword_t        i;
111     BYTE                 test[] = {0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18,
112                                    0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10,
113                                    0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,
114                                    0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
115     BN_VAR(tpmTemp, sizeof(test) * 8); // allocate some space for a test value
116 //
117     // Convert the test data to a bigNum
118     BnFromBytes(tpmTemp, test, sizeof(test));
119     // Convert the test data to an OpenSSL BIGNUM
120     BN_bin2bn(test, sizeof(test), osslTemp);
121     // Make sure the values are consistent
122     VERIFY(osslTemp->top == (int)tpmTemp->size);
123     for(i = 0; i < tpmTemp->size; i++)
124         VERIFY(osslTemp->d[i] == tpmTemp->d[i]);
125     OSSL_LEAVE();
126     return 1;
127 Error:
128     return 0;
129 }
```

```
130 #endif
```

B.2.3.2.3.6. BnModMult()

This function does a modular multiply. It first does a multiply and then a divide and returns the remainder of the divide.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
131 LIB_EXPORT BOOL
132 BnModMult(
133     bigNum              result,
134     bigConst            op1,
135     bigConst            op2,
136     bigConst            modulus
137     )
138 {
139     OSSL_ENTER();
140     BOOL                OK = TRUE;
141     BIGNUM              *bnResult = BN_NEW();
142     BIGNUM              *bnTemp = BN_NEW();
143     BIG_INITIALIZED(bnOp1, op1);
144     BIG_INITIALIZED(bnOp2, op2);
145     BIG_INITIALIZED(bnMod, modulus);
146 //
147     VERIFY(BN_mul(bnTemp, bnOp1, bnOp2, CTX));
148     VERIFY(BN_div(NULL, bnResult, bnTemp, bnMod, CTX));
149     VERIFY(OsslToTpmBn(result, bnResult));
150     goto Exit;
151 Error:
152     OK = FALSE;
153 Exit:
154     OSSL_LEAVE();
155     return OK;
156 }
```

B.2.3.2.3.7. BnMult()

Multiplies two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
157 LIB_EXPORT BOOL
158 BnMult(
159     bigNum              result,
160     bigConst            multiplicand,
161     bigConst            multiplier
162     )
163 {
164     OSSL_ENTER();
165     BIGNUM              *bnTemp = BN_NEW();
166     BOOL                 OK = TRUE;
167     BIG_INITIALIZED(bnA, multiplicand);
168     BIG_INITIALIZED(bnB, multiplier);
169 //
```

```
170         VERIFY(BN_mul(bnTemp, bnA, bnB, CTX));
171         VERIFY(OsslToTpmBn(result, bnTemp));
172         goto Exit;
173     Error:
174         OK = FALSE;
175     Exit:
176         OSSL_LEAVE();
177         return OK;
178     }
```

B.2.3.2.3.8. 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

```
179     LIB_EXPORT BOOL
180     BnDiv(
181         bigNum               quotient,
182         bigNum               remainder,
183         bigConst             dividend,
184         bigConst             divisor
185         )
186     {
187         OSSL_ENTER();
188         BIGNUM              *bnQ = BN_NEW();
189         BIGNUM              *bnR = BN_NEW();
190         BOOL                 OK = TRUE;
191         BIG_INITIALIZED(bnDend, dividend);
192         BIG_INITIALIZED(bnSor, divisor);
193     //
194         if(BnEqualZero(divisor))
195             FAIL(FATAL_ERROR_DIVIDE_ZERO);
196         VERIFY(BN_div(bnQ, bnR, bnDend, bnSor, CTX));
197         VERIFY(OsslToTpmBn(quotient, bnQ));
198         VERIFY(OsslToTpmBn(remainder, bnR));
199         DEBUG_PRINT("In BnDiv:\n");
200         BIGNUM_PRINT("   bnDividend: ", bnDend, TRUE);
201         BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
202         BIGNUM_PRINT("   bnQuotient: ", bnQ, TRUE);
203         BIGNUM_PRINT("  bnRemainder: ", bnR, TRUE);
204         goto Exit;
205     Error:
206         OK = FALSE;
207     Exit:
208         OSSL_LEAVE();
209         return OK;
210     }
211     #if ALG_RSA
```

B.2.3.2.3.9. BnGcd()

Get the greatest common divisor of two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
212  LIB_EXPORT BOOL
213  BnGcd(
214      bigNum      gcd,            // OUT: the common divisor
215      bigConst    number1,        // IN:
216      bigConst    number2         // IN:
217      )
218  {
219      OSSL_ENTER();
220      BIGNUM              *bnGcd = BN_NEW();
221      BOOL                 OK = TRUE;
222      BIG_INITIALIZED(bn1, number1);
223      BIG_INITIALIZED(bn2, number2);
224  //
225      VERIFY(BN_gcd(bnGcd, bn1, bn2, CTX));
226      VERIFY(OsslToTpmBn(gcd, bnGcd));
227      goto Exit;
228  Error:
229      OK = FALSE;
230  Exit:
231      OSSL_LEAVE();
232      return OK;
233  }
```

B.2.3.2.3.10. 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

```
234  LIB_EXPORT BOOL
235  BnModExp(
236      bigNum              result,         // OUT: the result
237      bigConst            number,         // IN: number to exponentiate
238      bigConst            exponent,       // IN:
239      bigConst            modulus         // IN:
240      )
241  {
242      OSSL_ENTER();
243      BIGNUM              *bnResult = BN_NEW();
244      BOOL                 OK = TRUE;
245      BIG_INITIALIZED(bnN, number);
246      BIG_INITIALIZED(bnE, exponent);
247      BIG_INITIALIZED(bnM, modulus);
248  //
249      VERIFY(BN_mod_exp(bnResult, bnN, bnE, bnM, CTX));
250      VERIFY(OsslToTpmBn(result, bnResult));
251      goto Exit;
252  Error:
253      OK = FALSE;
254  Exit:
255      OSSL_LEAVE();
256      return OK;
257  }
```

B.2.3.2.3.11. BnModInverse()

Modular multiplicative inverse

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
258  LIB_EXPORT BOOL
259  BnModInverse(
260      bigNum               result,
261      bigConst             number,
262      bigConst             modulus
263      )
264  {
265      OSSL_ENTER();
266      BIGNUM              *bnResult = BN_NEW();
267      BOOL                 OK = TRUE;
268      BIG_INITIALIZED(bnN, number);
269      BIG_INITIALIZED(bnM, modulus);
270  //
271      VERIFY(BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
272      VERIFY(OsslToTpmBn(result, bnResult));
273      goto Exit;
274  Error:
275      OK = FALSE;
276  Exit:
277      OSSL_LEAVE();
278      return OK;
279  }
280  #endif // ALG_RSA
281  #if ALG_ECC
```

B.2.3.2.3.12. 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

```
282  static BOOL
283  PointFromOssl(
284      bigPoint         pOut,       // OUT: resulting point
285      EC_POINT        *pIn,        // IN: the point to return
286      bigCurve         E           // IN: the curve
287      )
288  {
289      BIGNUM          *x = NULL;
290      BIGNUM          *y = NULL;
291      BOOL             OK;
292      BN_CTX_start(E->CTX);
293  //
294      x = BN_CTX_get(E->CTX);
295      y = BN_CTX_get(E->CTX);
296
297      if(y == NULL)
298          FAIL(FATAL_ERROR_ALLOCATION);
299      // If this returns false, then the point is at infinity
300      OK = EC_POINT_get_affine_coordinates(E->G, pIn, x, y, E->CTX);
```

```
301         if(OK)
302         {
303             OsslToTpmBn(pOut->x, x);
304             OsslToTpmBn(pOut->y, y);
305             BnSetWord(pOut->z, 1);
306         }
307         else
308             BnSetWord(pOut->z, 0);
309         BN_CTX_end(E->CTX);
310         return OK;
311     }
```

B.2.3.2.3.13. EcPointInitialized()

Allocate and initialize a point.

```
312     static EC_POINT *
313     EcPointInitialized(
314         pointConst          initializer,
315         bigCurve            E
316         )
317     {
318         EC_POINT            *P = NULL;
319
320         if(initializer != NULL)
321         {
322             BIG_INITIALIZED(bnX, initializer->x);
323             BIG_INITIALIZED(bnY, initializer->y);
324             P = EC_POINT_new(E->G);
325             if(E == NULL)
326                 FAIL(FATAL_ERROR_ALLOCATION);
327             if(!EC_POINT_set_affine_coordinates(E->G, P, bnX, bnY, E->CTX))
328                 P = NULL;
329         }
330         return P;
331     }
```

B.2.3.2.3.14. BnCurveInitialize()

This function initializes the OpenSSL curve information structure. This structure points to the TPM-defined values for the curve, to the context for the number values in the frame, and to the OpenSSL-defined group values.

Return Value	Meaning
NULL	the TPM_ECC_CURVE is not valid or there was a problem in in initializing the curve data
non-NULL	points to *E*

```
332     LIB_EXPORT bigCurve
333     BnCurveInitialize(
334         bigCurve            E,              // IN: curve structure to initialize
335         TPM_ECC_CURVE       curveId         // IN: curve identifier
336     )
337     {
338         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
339         if(C == NULL)
340             E = NULL;
341         if(E != NULL)
342         {
343             // This creates the OpenSSL memory context that stays in effect as long as the
344             // curve (E) is defined.
```

```
345             OSSL_ENTER();                       // if the allocation fails, the TPM fails
346             EC_POINT                *P = NULL;
347             BIG_INITIALIZED(bnP, C->prime);
348             BIG_INITIALIZED(bnA, C->a);
349             BIG_INITIALIZED(bnB, C->b);
350             BIG_INITIALIZED(bnX, C->base.x);
351             BIG_INITIALIZED(bnY, C->base.y);
352             BIG_INITIALIZED(bnN, C->order);
353             BIG_INITIALIZED(bnH, C->h);
354         //
355             E->C = C;
356             E->CTX = CTX;
357
358             // initialize EC group, associate a generator point and initialize the point
359             // from the parameter data
360             // Create a group structure
361             E->G = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX);
362             VERIFY(E->G != NULL);
363
364             // Allocate a point in the group that will be used in setting the
365             // generator. This is not needed after the generator is set.
366             P = EC_POINT_new(E->G);
367             VERIFY(P != NULL);
368
369             // Need to use this in case Montgomery method is being used
370             VERIFY(EC_POINT_set_affine_coordinates(E->G, P, bnX, bnY, CTX));
371             // Now set the generator
372             VERIFY(EC_GROUP_set_generator(E->G, P, bnN, bnH));
373
374             EC_POINT_free(P);
375             goto Exit;
376     Error:
377             EC_POINT_free(P);
378             BnCurveFree(E);
379             E = NULL;
380         }
381     Exit:
382         return E;
383     }
```

B.2.3.2.3.15. BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```
384     LIB_EXPORT void
385     BnCurveFree(
386         bigCurve                    E
387     )
388     {
389         if(E)
390         {
391             EC_GROUP_free(E->G);
392             OsslContextLeave(E->CTX);
393         }
394     }
```

B.2.3.2.3.16. 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

```
LIB_EXPORT BOOL
BnEccModMult(
    bigPoint             R,          // OUT: computed point
    pointConst           S,          // IN: point to multiply by 'd' (optional)
    bigConst             d,          // IN: scalar for [d]S
    bigCurve             E
    )
{
    EC_POINT            *pR = EC_POINT_new(E->G);
    EC_POINT            *pS = EcPointInitialized(S, E);
    BIG_INITIALIZED(bnD, d);

    if(S == NULL)
        EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
    else
        EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
    PointFromOssl(R, pR, E);
    EC_POINT_free(pR);
    EC_POINT_free(pS);
    return !BnEqualZero(R->z);
}
```

B.2.3.2.3.17. 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

```
LIB_EXPORT BOOL
BnEccModMult2(
    bigPoint             R,          // OUT: computed point
    pointConst           S,          // IN: optional point
    bigConst             d,          // IN: scalar for [d]S or [d]G
    pointConst           Q,          // IN: second point
    bigConst             u,          // IN: second scalar
    bigCurve             E           // IN: curve
    )
{
    EC_POINT            *pR = EC_POINT_new(E->G);
    EC_POINT            *pS = EcPointInitialized(S, E);
    BIG_INITIALIZED(bnD, d);
    EC_POINT            *pQ = EcPointInitialized(Q, E);
    BIG_INITIALIZED(bnU, u);

    if(S == NULL || S == (pointConst)&(AccessCurveData(E)->base))
        EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
    else
    {
        const EC_POINT        *points[2];
        const BIGNUM          *scalars[2];
        points[0] = pS;
        points[1] = pQ;
        scalars[0] = bnD;
        scalars[1] = bnU;
```

```
442             EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
443         }
444         PointFromOssl(R, pR, E);
445         EC_POINT_free(pR);
446         EC_POINT_free(pS);
447         EC_POINT_free(pQ);
448         return !BnEqualZero(R->z);
449     }
```

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

```
450     LIB_EXPORT BOOL
451     BnEccAdd(
452         bigPoint             R,          // OUT: computed point
453         pointConst           S,          // IN: point to multiply by 'd'
454         pointConst           Q,          // IN: second point
455         bigCurve             E           // IN: curve
456         )
457     {
458         EC_POINT            *pR = EC_POINT_new(E->G);
459         EC_POINT            *pS = EcPointInitialized(S, E);
460         EC_POINT            *pQ = EcPointInitialized(Q, E);
461     //
462         EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
463
464         PointFromOssl(R, pR, E);
465         EC_POINT_free(pR);
466         EC_POINT_free(pS);
467         EC_POINT_free(pQ);
468         return !BnEqualZero(R->z);
469     }
470     #endif // ALG_ECC
471     #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 defined(HASH_LIB_OSSL) || defined(MATH_LIB_OSSL) || defined(SYM_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
    )
{
    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              *CTX = BN_CTX_new();
//
    return OsslPushContext(CTX);
}
```

B.2.3.3.2.3. OsslContextLeave()

This is the companion function to OsslContextEnter().

```
void
OsslContextLeave(
    BN_CTX          *CTX
    )
{
    OsslPopContext(CTX);
    BN_CTX_free(CTX);
}
```

B.2.3.3.2.4. OsslPushContext()

This function is used to create a frame in a context. All values allocated within this context after the frame is started will be automatically freed when the context (OsslPopContext()

```
28  BN_CTX *
29  OsslPushContext(
30      BN_CTX          *CTX
31      )
32  {
33      if(CTX == NULL)
34          FAIL(FATAL_ERROR_ALLOCATION);
35      BN_CTX_start(CTX);
36      return CTX;
37  }
```

B.2.3.3.2.5. OsslPopContext()

This is the companion function to OsslPushContext().

```
38  void
39  OsslPopContext(
40      BN_CTX          *CTX
41      )
42  {
43      // BN_CTX_end can't be called with NULL. It will blow up.
44      if(CTX != NULL)
45          BN_CTX_end(CTX);
46  }
47  #endif // HASH_LIB_OSSL || MATH_LIB_OSSL || SYM_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

```
1   #include "Platform.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

```
2   LIB_EXPORT int
3   _plat__IsCanceled(
4       void
5       )
6   {
7       // return cancel flag
8       return s_isCanceled;
9   }
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```
10  LIB_EXPORT void
11  _plat__SetCancel(
12      void
13      )
14  {
15      s_isCanceled = TRUE;
16      return;
17  }
```


C.2.3.3. _plat__ClearCancel()

Clear cancel flag

```
18  LIB_EXPORT void
19  _plat__ClearCancel(
20      void
21      )
22  {
23      s_isCanceled = FALSE;
24      return;
25  }
```

C.3 Clock.c

C.3.1. 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

```
#include <assert.h>
#include "Platform.h"
#include "TpmFail_fp.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.

```
LIB_EXPORT void
_plat__TimerReset(
    void
    )
{
    s_lastSystemTime = 0;
    s_tpmTime = 0;
    s_adjustRate = CLOCK_NOMINAL;
    s_timerReset = TRUE;
    s_timerStopped = TRUE;
    return;
}
```

C.3.3.3. _plat__TimerRestart()

This function should be called in order to simulate the restart of the timer should it be stopped while power is still applied.

```
LIB_EXPORT void
_plat__TimerRestart(
    void
    )
{
    s_timerStopped = TRUE;
    return;
}
```


C.3.4. Functions Used by TPM

C.3.4.1. Introduction

These functions are called by the TPM code. They should be replaced by appropriated hardware functions.

```
24  #include <time.h>
25  clock_t     debugTime;
26
27  //*** _plat__RealTime()
28  // This is another, probably futile, attempt to define a portable function
29  // that will return a 64-bit clock value that has mSec resolution.
30  LIB_EXPORT uint64_t
31  _plat__RealTime(
32      void
33  )
34  {
35      clock64_t           time;
36  #ifdef _MSC_VER
37      struct _timeb       sysTime;
38  //
39      _ftime_s(&sysTime);
40      time = (clock64_t)(sysTime.time) * 1000 + sysTime.millitm;
41      // set the time back by one hour if daylight savings
42      if(sysTime.dstflag)
43          time -= 1000 * 60 * 60;  // mSec/sec * sec/min * min/hour = ms/hour
44  #else
45      // hopefully, this will work with most UNIX systems
46      struct timespec     systime;
47  //
48      clock_gettime(CLOCK_MONOTONIC, &systime);
49      time = (clock64_t)systime.tv_sec * 1000 + (systime.tv_nsec / 1000000);
50  #endif
51      return time;
52  }
```

C.3.4.2. _plat__TimerRead()

This function provides access to the tick timer of the platform. The TPM code uses this value to drive the TPM Clock.

The tick timer is supposed to run when power is applied to the device. This timer should not be reset by time events including _TPM_Init(). It should only be reset when TPM power is re-applied.

If the TPM is run in a protected environment, that environment may provide the tick time to the TPM as long as the time provided by the environment is not allowed to go backwards. If the time provided by the system can go backwards during a power discontinuity, then the _plat__Signal_PowerOn() should call _plat__TimerReset().

```
53  LIB_EXPORT uint64_t
54  _plat__TimerRead(
55      void
56      )
57  {
58  #ifdef HARDWARE_CLOCK
59  #error      "need a defintion for reading the hardware clock"
60      return HARDWARE_CLOCK
61  #else
62      clock64_t         timeDiff;
63      clock64_t         adjustedTimeDiff;
64      clock64_t         timeNow;
```

```
65      clock64_t         readjustedTimeDiff;
66
67      // This produces a timeNow that is basically locked to the system clock.
68      timeNow = _plat__RealTime();
69
70      // if this hasn't been initialized, initialize it
71      if(s_lastSystemTime == 0)
72      {
73          s_lastSystemTime = timeNow;
74          debugTime = clock();
75          s_lastReportedTime = 0;
76          s_realTimePrevious = 0;
77      }
78      // The system time can bounce around and that's OK as long as we don't allow
79      // time to go backwards. When the time does appear to go backwards, set
80      // lastSystemTime to be the new value and then update the reported time.
81      if(timeNow < s_lastReportedTime)
82          s_lastSystemTime = timeNow;
83      s_lastReportedTime = s_lastReportedTime + timeNow - s_lastSystemTime;
84      s_lastSystemTime = timeNow;
85      timeNow = s_lastReportedTime;
86
87      // The code above produces a timeNow that is similar to the value returned
88      // by Clock(). The difference is that timeNow does not max out, and it is
89      // at a ms. rate rather than at a CLOCKS_PER_SEC rate. The code below
90      // uses that value and does the rate adjustment on the time value.
91      // If there is no difference in time, then skip all the computations
92      if(s_realTimePrevious >= timeNow)
93          return s_tpmTime;
94      // Compute the amount of time since the last update of the system clock
95      timeDiff = timeNow - s_realTimePrevious;
96
97      // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
98      adjustedTimeDiff = (timeDiff * CLOCK_NOMINAL) / ((uint64_t)s_adjustRate);
99
100     // update the TPM time with the adjusted timeDiff
101     s_tpmTime += (clock64_t)adjustedTimeDiff;
102
103     // Might have some rounding error that would loose CLOCKS. See what is not
104     // being used. As mentioned above, this could result in putting back more than
105     // is taken out. Here, we are trying to recreate timeDiff.
106     readjustedTimeDiff = (adjustedTimeDiff * (uint64_t)s_adjustRate )
107                                 / CLOCK_NOMINAL;
108
109     // adjusted is now converted back to being the amount we should advance the
110     // previous sampled time. It should always be less than or equal to timeDiff.
111     // That is, we could not have use more time than we started with.
112     s_realTimePrevious = s_realTimePrevious + readjustedTimeDiff;
113
114 #ifdef  DEBUGGING_TIME
115     // Put this in so that TPM time will pass much faster than real time when
116     // doing debug.
117     // A value of 1000 for DEBUG_TIME_MULTIPLER will make each ms into a second
118     // A good value might be 100
119     return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
120 #endif
121     return s_tpmTime;
122 #endif
123 }
```

C.3.4.3. _plat__TimerWasReset()

This function is used to interrogate the flag indicating if the tick timer has been reset.

If the *resetFlag* parameter is SET, then the flag will be CLEAR before the function returns.

```
124 LIB_EXPORT int
125 _plat__TimerWasReset(
126    void
127     )
128 {
129     int          retVal = s_timerReset;
130     s_timerReset = FALSE;
131     return retVal;
132 }
```

C.3.4.4. _plat__TimerWasStopped()

This function is used to interrogate the flag indicating if the tick timer has been stopped. If so, this is typically a reason to roll the nonce.

This function will CLEAR the *s_timerStopped* flag before returning. This provides functionality that is similar to status register that is cleared when read. This is the model used here because it is the one that has the most impact on the TPM code as the flag can only be accessed by one entity in the TPM. Any other implementation of the hardware can be made to look like a read-once register.

```
133 LIB_EXPORT int
134 _plat__TimerWasStopped(
135     void
136     )
137 {
138     int          retVal = s_timerStopped;
139     s_timerStopped = FALSE;
140     return retVal;
141 }
```

C.3.4.5. _plat__ClockAdjustRate()

Adjust the clock rate

```
142 LIB_EXPORT void
143 _plat__ClockAdjustRate(
144     int              adjust         // IN: the adjust number.  It could be positive
145                                     //     or negative
146     )
147 {
148     // We expect the caller should only use a fixed set of constant values to
149     // adjust the rate
150     switch(adjust)
151     {
152         case CLOCK_ADJUST_COARSE:
153             s_adjustRate += CLOCK_ADJUST_COARSE;
154             break;
155         case -CLOCK_ADJUST_COARSE:
156             s_adjustRate -= CLOCK_ADJUST_COARSE;
157             break;
158         case CLOCK_ADJUST_MEDIUM:
159             s_adjustRate += CLOCK_ADJUST_MEDIUM;
160             break;
161         case -CLOCK_ADJUST_MEDIUM:
162             s_adjustRate -= CLOCK_ADJUST_MEDIUM;
163             break;
164         case CLOCK_ADJUST_FINE:
165             s_adjustRate += CLOCK_ADJUST_FINE;
166             break;
167         case -CLOCK_ADJUST_FINE:
168             s_adjustRate -= CLOCK_ADJUST_FINE;
169             break;
```

```
170             default:
171                 // ignore any other values;
172                 break;
173         }
174 
175         if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
176             s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
177         if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
178             s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
179 
180         return;
181     }
```

C.4 Entropy.c

C.4.1. Includes and Local Values

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include "Platform.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;

//** Functions

//*** 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.4.1.1. _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)

```
38  LIB_EXPORT int32_t
39  _plat__GetEntropy(
40      unsigned char       *entropy,           // output buffer
41      uint32_t             amount             // amount requested
42  )
43  {
44      uint32_t            rndNum;
45      int32_t             ret;
46  //
47      if(amount == 0)
48      {
49          // Seed the platform entropy source if the entropy source is software. There
50          // is no reason to put a guard macro (#if or #ifdef) around this code because
51          // this code would not be here if someone was changing it for a system with
52          // actual hardware.
53          //
54          // NOTE 1: The following command does not provide proper cryptographic
55          // entropy. Its primary purpose to make sure that different instances of the
56          // simulator, possibly started by a script on the same machine, are seeded
57          // differently. Vendors of the actual TPMs need to ensure availability of
58          // proper entropy using their platform-specific means.
59          //
60          // NOTE 2: In debug builds by default the reference implementation will seed
61          // its RNG deterministically (without using any platform provided randomness).
62          // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.
63  #ifdef _MSC_VER
64          srand((unsigned)_plat__RealTime() ^ _getpid());
65  #else
66          srand((unsigned)_plat__RealTime() ^ getpid());
67  #endif
68          lastEntropy = rand32();
69          ret = 0;
70      }
71      else
72      {
73          rndNum = rand32();
74          if(rndNum == lastEntropy)
75          {
76              ret = -1;
77          }
78          else
79          {
80              lastEntropy = rndNum;
81              // Each process will have its random number generator initialized
82              // according to the process id and the initialization time. This is not a
83              // lot of entropy so, to add a bit more, XOR the current time value into
84              // the returned entropy value.
85              // NOTE: the reason for including the time here rather than have it in
86              // in the value assigned to lastEntropy is that rand() could be broken and
87              // using the time would in the lastEntropy value would hide this.
88              rndNum ^= (uint32_t)_plat__RealTime();
89
90              // Only provide entropy 32 bits at a time to test the ability
91              // of the caller to deal with partial results.
92              ret = MIN(amount, sizeof(rndNum));
93              memcpy(entropy, &rndNum, ret);
94          }
95      }
96      return ret;
```

```
97 }
```

C.5 LocalityPlat.c

C.5.1. Includes

```
#include "Platform.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

```
#include <memory.h>
#include <string.h>
#include <assert.h>
#include "Platform.h"
#if FILE_BACKED_NV
#   include         <stdio.h>
FILE                *s_NvFile = NULL;
int                  s_NeedsManufacture = FALSE;
#endif

//**Functions

#if FILE_BACKED_NV

//*** NvFileOpen()
// This function opens the file used to hold the NV image.
//  Return Type: int
//  >= 0        success
//  -1          error
static int
NvFileOpen(
    const char      *mode
)
{
#if defined(NV_FILE_PATH)
#   define TO_STRING(s) TO_STRING_IMPL(s)
#   define TO_STRING_IMPL(s) #s
    const char* s_NvFilePath = TO_STRING(NV_FILE_PATH);
#   undef TO_STRING
#   undef TO_STRING_IMPL
#else
    const char* s_NvFilePath = "NVChip";
#endif

    // Try to open an exist NVChip file for read/write
#   if defined _MSC_VER && 1
    if(fopen_s(&s_NvFile, s_NvFilePath, mode) != 0)
        s_NvFile = NULL;
#   else
    s_NvFile = fopen(s_NvFilePath, mode);
#   endif
    return (s_NvFile == NULL) ? -1 : 0;
}
```

C.6.2.1. NvFileCommit()

Write all of the contents of the NV image to a file.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
44  static int
45  NvFileCommit(
46      void
47  )
48  {
49      int         OK;
50      // If NV file is not available, return failure
51      if(s_NvFile == NULL)
52          return 1;
53      // Write RAM data to NV
54      fseek(s_NvFile, 0, SEEK_SET);
55      OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
56      OK = OK && (0 == fflush(s_NvFile));
57      assert(OK);
58      return OK;
59  }
```

C.6.2.2. 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.

```
60  static long
61  NvFileSize(
62      int         leaveAt
63  )
64  {
65      long    fileSize;
66      long    filePos = ftell(s_NvFile);
67  //
68      assert(NULL != s_NvFile);
69
70      fseek(s_NvFile, 0, SEEK_END);
71      fileSize = ftell(s_NvFile);
72      switch(leaveAt)
73      {
74          case SEEK_SET:
75              filePos = 0;
76          case SEEK_CUR:
77              fseek(s_NvFile, filePos, SEEK_SET);
78              break;
79          case SEEK_END:
80              break;
81          default:
82              assert(FALSE);
83              break;
84      }
85      return fileSize;
86  }
87  #endif
```

C.6.2.3. _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

```
88  LIB_EXPORT void
89  _plat__NvErrors(
90      int                 recoverable,
91      int                 unrecoverable
92      )
93  {
94      s_NV_unrecoverable = unrecoverable;
95      s_NV_recoverable = recoverable;
96  }
```

C.6.2.4. _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

```
97  LIB_EXPORT int
98  _plat__NVEnable(
99      void            *platParameter  // IN: platform specific parameters
100     )
101 {
102     NOT_REFERENCED(platParameter);          // to keep compiler quiet
103 //
104     // Start assuming everything is OK
105     s_NV_unrecoverable = FALSE;
106     s_NV_recoverable = FALSE;
107 #if FILE_BACKED_NV
108     if(s_NvFile != NULL)
109         return 0;
110     // Initialize all the bytes in the ram copy of the NV
111     _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
112
113     // If the file exists
114     if(NvFileOpen("r+b") >= 0)
115     {
116         long    fileSize = NvFileSize(SEEK_SET);    // get the file size and leave the
117                                                     // file pointer at the start
118 //
119         // If the size is right, read the data
120         if (NV_MEMORY_SIZE == fileSize)
121         {
122             s_NeedsManufacture =
123                 fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile) != NV_MEMORY_SIZE;
124         }
125         else
126         {
127             NvFileCommit();     // for any other size, initialize it
128             s_NeedsManufacture = TRUE;
129         }
130     }
```

```
131         // If NVChip file does not exist, try to create it for read/write.
132         else if(NvFileOpen("w+b") >= 0)
133         {
134             NvFileCommit();             // Initialize the file
135             s_NeedsManufacture = TRUE;
136         }
137         assert(NULL != s_NvFile);       // Just in case we are broken for some reason.
138     #endif
139         // NV contents have been initialized and the error checks have been performed. For
140         // simulation purposes, use the signaling interface to indicate if an error is
141         // to be simulated and the type of the error.
142         if(s_NV_unrecoverable)
143             return -1;
144         return s_NV_recoverable;
145     }
```

C.6.2.5. _plat__NVDisable()

Disable NV memory

```
146     LIB_EXPORT void
147     _plat__NVDisable(
148         int              delete           // IN: If TRUE, delete the NV contents.
149         )
150     {
151     #if  FILE_BACKED_NV
152         if(NULL != s_NvFile)
153         {
154             fclose(s_NvFile);    // Close NV file
155             // Alternative to deleting the file is to set its size to 0. This will not
156             // match the NV size so the TPM will need to be remanufactured.
157             if(delete)
158             {
159                 // Open for writing at the start. Sets the size to zero.
160                 if(NvFileOpen("w") >= 0)
161                 {
162                     fflush(s_NvFile);
163                     fclose(s_NvFile);
164                 }
165             }
166         }
167         s_NvFile = NULL;        // Set file handle to NULL
168     #endif
169         return;
170     }
```

C.6.2.6. _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

```
171     LIB_EXPORT int
172     _plat__IsNvAvailable(
173         void
174         )
175     {
```

```
176     int         retVal = 0;
177     // NV is not available if the TPM is in failure mode
178     if(!s_NvIsAvailable)
179         retVal = 1;
180 #if FILE_BACKED_NV
181     else
182         retVal = (s_NvFile == NULL);
183 #endif
184     return retVal;
185 }
```

C.6.2.7. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
186 LIB_EXPORT void
187 _plat__NvMemoryRead(
188     unsigned int     startOffset,   // IN: read start
189     unsigned int     size,          // IN: size of bytes to read
190     void            *data           // OUT: data buffer
191     )
192 {
193     assert(startOffset + size <= NV_MEMORY_SIZE);
194     memcpy(data, &s_NV[startOffset], size);     // Copy data from RAM
195     return;
196 }
```

C.6.2.8. _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

```
197 LIB_EXPORT int
198 _plat__NvIsDifferent(
199     unsigned int     startOffset,   // IN: read start
200     unsigned int     size,          // IN: size of bytes to read
201     void            *data           // IN: data buffer
202     )
203 {
204     return (memcmp(&s_NV[startOffset], data, size) != 0);
205 }
```

C.6.2.9. _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.

```
206 LIB_EXPORT int
207 _plat__NvMemoryWrite(
208     unsigned int     startOffset,   // IN: write start
209     unsigned int     size,          // IN: size of bytes to write
210     void            *data           // OUT: data buffer
```

```
211         )
212     {
213         if(startOffset + size <= NV_MEMORY_SIZE)
214         {
215             memcpy(&s_NV[startOffset], data, size);     // Copy the data to the NV image
216             return TRUE;
217         }
218         return FALSE;
219     }
```

C.6.2.10. _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.

```
220     LIB_EXPORT void
221     _plat__NvMemoryClear(
222         unsigned int     start,         // IN: clear start
223         unsigned int     size           // IN: number of bytes to clear
224         )
225     {
226         assert(start + size <= NV_MEMORY_SIZE);
227         // In this implementation, assume that the erase value for NV is all 1s
228         memset(&s_NV[start], 0xff, size);
229     }
```

C.6.2.11. _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

```
230     LIB_EXPORT void
231     _plat__NvMemoryMove(
232         unsigned int     sourceOffset,  // IN: source offset
233         unsigned int     destOffset,    // IN: destination offset
234         unsigned int     size           // IN: size of data being moved
235         )
236     {
237         assert(sourceOffset + size <= NV_MEMORY_SIZE);
238         assert(destOffset + size <= NV_MEMORY_SIZE);
239         memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);      // Move data in RAM
240         return;
241     }
```

C.6.2.12. _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

```
242     LIB_EXPORT int
243     _plat__NvCommit(
244         void
245         )
246     {
247     #if FILE_BACKED_NV
```

```
248         return (NvFileCommit() ? 0 : 1);
249     #else
250         return 0;
251     #endif
252     }
```

C.6.2.13. _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

```
253     LIB_EXPORT void
254     _plat__SetNvAvail(
255         void
256         )
257     {
258         s_NvIsAvailable = TRUE;
259         return;
260     }
```

C.6.2.14. _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

```
261     LIB_EXPORT void
262     _plat__ClearNvAvail(
263         void
264         )
265     {
266         s_NvIsAvailable = FALSE;
267         return;
268     }
```

C.6.2.15. _plat__NVNeedsManufacture()

This function is used by the simulator to determine when the TPM's NV state needs to be manufactured.

```
269     LIB_EXPORT int
270     _plat__NVNeedsManufacture(
271         void
272         )
273     {
274     #if FILE_BACKED_NV
275         return s_NeedsManufacture;
276     #else
277         return FALSE;
278     #endif
279     }
```

C.7 PowerPlat.c

C.7.1. Includes and Function Prototypes

```
#include    "Platform.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
    )
{
    int         retVal = s_powerLost;
    s_powerLost = FALSE;
    return retVal;
}
```

C.7.2.3. _plat_Signal_Reset()

This a TPM reset without a power loss.

```
LIB_EXPORT int
_plat__Signal_Reset(
```

```
27      void
28      )
29  {
30      // Initialize locality
31      s_locality = 0;
32
33      // Command cancel
34      s_isCanceled = FALSE;
35
36      _TPM_Init();
37
38      // if we are doing reset but did not have a power failure, then we should
39      // not need to reload NV ...
40
41      return 0;
42  }
```

C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

```
43  LIB_EXPORT void
44  _plat__Signal_PowerOff(
45      void
46      )
47  {
48      // Prepare NV memory for power off
49      _plat__NVDisable(0);
50
51      // Disable tick ACT tick processing
52      _plat__ACT_EnableTicks(FALSE);
53
54      return;
55  }
```

C.8 PlatformData.h

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```
#ifndef _PLATFORM_DATA_H_
#define _PLATFORM_DATA_H_
#ifdef  _PLATFORM_DATA_C_
#define EXTERN
#else
#define EXTERN  extern
#endif
```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```
EXTERN int     s_isCanceled;

#ifndef HARDWARE_CLOCK
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.
EXTERN clock64_t       s_realTimePrevious;

// These values are used to try to synthesize a long lived version of clock().
EXTERN clock64_t        s_lastSystemTime;
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.
EXTERN clock64_t        s_tpmTime;
#endif // HARDWARE_CLOCK

// This value indicates that the timer was reset
EXTERN int               s_timerReset;
// This value indicates that the timer was stopped. It causes a clock discontinuity.
EXTERN int               s_timerStopped;

// 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
EXTERN uint64_t        s_initClock;

// This variable records the timer adjustment factor.
EXTERN unsigned int         s_adjustRate;

// For LocalityPlat.c
// Locality of current command
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.

#if (!defined VTPM) || ((VTPM != NO) && (VTPM != YES))
#   undef VTPM
#   define      VTPM            YES                 // Default: Either YES or NO
#endif
```

For a simulation, use a file to back up the NV

```
#if (!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))
#   undef   FILE_BACKED_NV
```

```
53  #   define  FILE_BACKED_NV          (VTPM && YES)     // Default: Either YES or NO
54  #endif
55  #if SIMULATION
56  #   undef       FILE_BACKED_NV
57  #   define      FILE_BACKED_NV          YES
58  #endif // SIMULATION
59  EXTERN unsigned char     s_NV[NV_MEMORY_SIZE];
60  EXTERN int               s_NvIsAvailable;
61  EXTERN int               s_NV_unrecoverable;
62  EXTERN int               s_NV_recoverable;
63
64  // For PPPlat.c
65  // Physical presence.  It is initialized to FALSE
66  EXTERN int               s_physicalPresence;
67
68  // From Power
69  EXTERN int               s_powerLost;
70
71  // For Entropy.c
72  EXTERN uint32_t          lastEntropy;
73
74  #define DEFINE_ACT(N)   EXTERN ACT_DATA ACT_##N;
75      FOR_EACH_ACT(DEFINE_ACT)
76  EXTERN int               actTicksAllowed;
77
78  #endif // _PLATFORM_DATA_H_
```

C.9 PlatformData.c

C.9.1. 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.9.2. Includes

```
1    #define _PLATFORM_DATA_C_
2    #include    "Platform.h"
```


C.10 PPPlat.c

C.10.1. Description

This module simulates the physical presence interface pins on the TPM.

C.10.2. Includes

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

C.10.3. Functions

C.10.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

Return Value	Meaning
TRUE(1)	if physical presence is signaled
FALSE(0)	if physical presence is not signaled

```
LIB_EXPORT int
_plat__PhysicalPresenceAsserted(
    void
    )
{
    // Do not know how to check physical presence without real hardware.
    // so always return TRUE;
    return s_physicalPresence;
}
```

C.10.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOn(
    void
    )
{
    s_physicalPresence = TRUE;
    return;
}
```

C.10.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOff(
    void
    )
{
    s_physicalPresence = FALSE;
    return;
}
```


C.11 RunCommand.c

C.11.1. 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.11.2. Includes and locals

```
#include "Platform.h"
#include <setjmp.h>
#include "ExecCommand_fp.h"
jmp_buf              s_jumpBuffer;

//** Functions

//***_plat__RunCommand()
// This version of RunCommand will set up a jum_buf and call ExecuteCommand(). If
// the command executes without failing, it will return and RunCommand will return.
// If there is a failure in the command, then _plat__Fail() is called and it will
// longjump back to RunCommand which will call ExecuteCommand again. However, this
// time, the TPM will be in failure mode so ExecuteCommand will simply build
// a failure response and return.
LIB_EXPORT void
_plat__RunCommand(
    uint32_t         requestSize,   // IN: command buffer size
    unsigned char   *request,       // IN: command buffer
    uint32_t        *responseSize,  // IN/OUT: response buffer size
    unsigned char   **response      // IN/OUT: response buffer
    )
{
    setjmp(s_jumpBuffer);
    ExecuteCommand(requestSize, request, responseSize, response);
}
```

C.11.2.1. _plat__Fail()

This is the platform depended failure exit for the TPM.

```
LIB_EXPORT NORETURN void
_plat__Fail(
    void
    )
{
    longjmp(&s_jumpBuffer[0], 1);
}
```

C.12 Unique.c

C.12.1. Introduction

In some implementations of the TPM, the hardware can provide a secret value to the TPM. This secret value is statistically unique to the instance of the TPM. Typical uses of this value are to provide personalization to the random number generation and as a shared secret between the TPM and the manufacturer.

C.12.2. Includes

```
1   #include "Platform.h"
2   const char notReallyUnique[] =
3   "This is not really a unique value. A real unique value should"
4   " be generated by the platform.";
5
6   //** _plat__GetUnique()
7   // This function is used to access the platform-specific unique value.
8   // This function places the unique value in the provided buffer ('b')
9   // and returns the number of bytes transferred. The function will not
10  // copy more data than 'bSize'.
11  // NOTE: If a platform unique value has unequal distribution of uniqueness
12  // and 'bSize' is smaller than the size of the unique value, the 'bSize'
13  // portion with the most uniqueness should be returned.
14  LIB_EXPORT uint32_t
15  _plat__GetUnique(
16      uint32_t             which,         // authorities (0) or details
17      uint32_t             bSize,         // size of the buffer
18      unsigned char       *b              // output buffer
19      )
20  {
21      const char          *from = notReallyUnique;
22      uint32_t             retVal = 0;
23
24      if(which == 0) // the authorities value
25      {
26          for(retVal = 0;
27          *from != 0 && retVal < bSize;
28              retVal++)
29          {
30              *b++ = *from++;
31          }
32      }
33      else
34      {
35  #define uSize  sizeof(notReallyUnique)
36          b = &b[((bSize < uSize) ? bSize : uSize) - 1];
37          for(retVal = 0;
38          *from != 0 && retVal < bSize;
39              retVal++)
40          {
41              *b-- = *from++;
42          }
43      }
44      return retVal;
45  }
```

C.13 DebugHelpers.c

C.13.1. 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.13.2. Includes and Local

```
#include <stdio.h>
#include <time.h>
#include "Platform.h"
#if CERTIFYX509_DEBUG
FILE            *fDebug = NULL;
const char      *debugFileName = "DebugFile.txt";

static FILE *
fileOpen(
    const char      *fn,
    const char      *mode
)
{
    FILE        *f;
#   if defined _MSC_VER
    if(fopen_s(&f, fn, mode) != 0)
        f = NULL;
#   else
    f = fopen(fn, "w");
#   endif
    return f;
}
```

C.13.2.1. DebugFileOpen()

This function opens the file used to hold the debug data.

Return Value	Meaning
0	success
0	error

```
int
DebugFileOpen(
    void
)
{
    time_t                  t = time(NULL);
//
    // Get current date and time.
#   if defined _MSC_VER
    char                timeString[100];
    ctime_s(timeString, (size_t)sizeof(timeString), &t);
#   else
    char                *timeString;
    timeString = ctime(&t);
#   endif
    // Try to open the debug file
    fDebug = fileOpen(debugFileName, "w");
    if(fDebug)
    {
```

```
42          fprintf(fDebug, "%s\n", timeString);
43          fclose(fDebug);
44          return 0;
45      }
46      return -1;
47  }
```

C.13.2.2. DebugFileClose()

```
48  void
49  DebugFileClose(
50      void
51  )
52  {
53      if(fDebug)
54          fclose(fDebug);
55  }
```

C.13.2.3. DebugDumpBuffer()

```
56  void
57  DebugDumpBuffer(
58      int             size,
59      unsigned char   *buf,
60      const char      *identifier
61  )
62  {
63      int             i;
64  //
65      FILE *f = fileOpen(debugFileName, "a");
66      if(!f)
67          return;
68      if(identifier)
69          fprintf(fDebug, "%s\n", identifier);
70      if(buf)
71      {
72          for(i = 0; i < size; i++)
73          {
74              if(((i % 16) == 0) && (i))
75                  fprintf(fDebug, "\n");
76              fprintf(fDebug, " %02X", buf[i]);
77          }
78          if((size % 16) != 0)
79              fprintf(fDebug, "\n");
80      }
81      fclose(f);
82  }
83  #endif // CERTIFYX509_DEBUG
```

C.14 Platform.h

```
#ifndef    _PLATFORM_H_
#define    _PLATFORM_H_
#include "TpmBuildSwitches.h"
#include "BaseTypes.h"
#include "TPMB.h"
#include "MinMax.h"
#include "TpmProfile.h"
#include "PlatformACT.h"
#include "PlatformClock.h"
#include "PlatformData.h"
#include "Platform_fp.h"
#endif  // _PLATFORM_H_
```

C.15 PlatformACT.h

This file contains the definitions for the ACT macros and data types used in the ACT implementation.

```
#ifndef _PLATFORM_ACT_H_
#define _PLATFORM_ACT_H_
typedef struct ACT_DATA
{
    uint32_t            remaining;
    uint32_t            newValue;
    uint8_t             signaled;
    uint8_t             pending;
    uint8_t             number;
} ACT_DATA, *P_ACT_DATA;
#if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
#   undef   RH_ACT_0
#   define  RH_ACT_0 NO
#   define IF_ACT_0_IMPLEMENTED(op)
#else
#   define IF_ACT_0_IMPLEMENTED(op) op(0)
#endif
#if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
#   undef   RH_ACT_1
#   define  RH_ACT_1 NO
#   define IF_ACT_1_IMPLEMENTED(op)
#else
#   define IF_ACT_1_IMPLEMENTED(op) op(1)
#endif
#if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
#   undef   RH_ACT_2
#   define  RH_ACT_2 NO
#   define IF_ACT_2_IMPLEMENTED(op)
#else
#   define IF_ACT_2_IMPLEMENTED(op) op(2)
#endif
#if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
#   undef   RH_ACT_3
#   define  RH_ACT_3 NO
#   define IF_ACT_3_IMPLEMENTED(op)
#else
#   define IF_ACT_3_IMPLEMENTED(op) op(3)
#endif
#if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
#   undef   RH_ACT_4
#   define  RH_ACT_4 NO
#   define IF_ACT_4_IMPLEMENTED(op)
#else
#   define IF_ACT_4_IMPLEMENTED(op) op(4)
#endif
#if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
#   undef   RH_ACT_5
#   define  RH_ACT_5 NO
#   define IF_ACT_5_IMPLEMENTED(op)
#else
#   define IF_ACT_5_IMPLEMENTED(op) op(5)
#endif
#if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
#   undef   RH_ACT_6
#   define  RH_ACT_6 NO
#   define IF_ACT_6_IMPLEMENTED(op)
#else
#   define IF_ACT_6_IMPLEMENTED(op) op(6)
#endif
#if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
#   undef   RH_ACT_7
```

```
62  #   define  RH_ACT_7 NO
63  #   define IF_ACT_7_IMPLEMENTED(op)
64  #else
65  #   define IF_ACT_7_IMPLEMENTED(op) op(7)
66  #endif
67  #if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
68  #   undef   RH_ACT_8
69  #   define  RH_ACT_8 NO
70  #   define IF_ACT_8_IMPLEMENTED(op)
71  #else
72  #   define IF_ACT_8_IMPLEMENTED(op) op(8)
73  #endif
74  #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
75  #   undef   RH_ACT_9
76  #   define  RH_ACT_9 NO
77  #   define IF_ACT_9_IMPLEMENTED(op)
78  #else
79  #   define IF_ACT_9_IMPLEMENTED(op) op(9)
80  #endif
81  #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
82  #   undef   RH_ACT_A
83  #   define  RH_ACT_A NO
84  #   define IF_ACT_A_IMPLEMENTED(op)
85  #else
86  #   define IF_ACT_A_IMPLEMENTED(op) op(A)
87  #endif
88  #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
89  #   undef   RH_ACT_B
90  #   define  RH_ACT_B NO
91  #   define IF_ACT_B_IMPLEMENTED(op)
92  #else
93  #   define IF_ACT_B_IMPLEMENTED(op) op(B)
94  #endif
95  #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
96  #   undef   RH_ACT_C
97  #   define  RH_ACT_C NO
98  #   define IF_ACT_C_IMPLEMENTED(op)
99  #else
100 #   define IF_ACT_C_IMPLEMENTED(op) op(C)
101 #endif
102 #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
103 #   undef   RH_ACT_D
104 #   define  RH_ACT_D NO
105 #   define IF_ACT_D_IMPLEMENTED(op)
106 #else
107 #   define IF_ACT_D_IMPLEMENTED(op) op(D)
108 #endif
109 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
110 #   undef   RH_ACT_E
111 #   define  RH_ACT_E NO
112 #   define IF_ACT_E_IMPLEMENTED(op)
113 #else
114 #   define IF_ACT_E_IMPLEMENTED(op) op(E)
115 #endif
116 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
117 #   undef   RH_ACT_F
118 #   define  RH_ACT_F NO
119 #   define IF_ACT_F_IMPLEMENTED(op)
120 #else
121 #   define IF_ACT_F_IMPLEMENTED(op) op(F)
122 #endif
123 #define FOR_EACH_ACT(op)                                    \
124     IF_ACT_0_IMPLEMENTED(op)                                \
125     IF_ACT_1_IMPLEMENTED(op)                                \
126     IF_ACT_2_IMPLEMENTED(op)                                \
127     IF_ACT_3_IMPLEMENTED(op)                                \
```

```
128         IF_ACT_4_IMPLEMENTED(op)                                    \
129         IF_ACT_5_IMPLEMENTED(op)                                    \
130         IF_ACT_6_IMPLEMENTED(op)                                    \
131         IF_ACT_7_IMPLEMENTED(op)                                    \
132         IF_ACT_8_IMPLEMENTED(op)                                    \
133         IF_ACT_9_IMPLEMENTED(op)                                    \
134         IF_ACT_A_IMPLEMENTED(op)                                    \
135         IF_ACT_B_IMPLEMENTED(op)                                    \
136         IF_ACT_C_IMPLEMENTED(op)                                    \
137         IF_ACT_D_IMPLEMENTED(op)                                    \
138         IF_ACT_E_IMPLEMENTED(op)                                    \
139         IF_ACT_F_IMPLEMENTED(op)
140     #endif // _PLATFORM_ACT_H_
```

C.16 PlatformACT.c

C.16.1. Includes

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

C.16.2. Functions

C.16.2.1. ActSignal()

Function called when there is an ACT event to signal or unsignal

```
static void
ActSignal(
    P_ACT_DATA          actData,
    int                 on
)
{
    if(actData == NULL)
        return;
    // If this is to turn a signal on, don't do anything if it is already on. If this
    // is to turn the signal off, do it anyway because this might be for
    // initialization.
    if(on && (actData->signaled == TRUE))
        return;
    actData->signaled = (uint8_t)on;

    // If there is an action, then replace the "Do something" with the correct action.
    // It should test 'on' to see if it is turning the signal on or off.
    switch(actData->number)
    {
#if RH_ACT_0
        case 0: // Do something
            return;
#endif
#if RH_ACT_1
        case 1: // Do something
            return;
#endif
#if RH_ACT_2
        case 2: // Do something
            return;
#endif
#if RH_ACT_3
        case 3: // Do something
            return;
#endif
#if RH_ACT_4
        case 4: // Do something
            return;
#endif
#if RH_ACT_5
        case 5: // Do something
            return;
#endif
#if RH_ACT_6
        case 6: // Do something
            return;
#endif
#if RH_ACT_7
        case 7: // Do something
            return;
```

```
52  #endif
53  #if RH_ACT_8
54          case 8: // Do something
55              return;
56  #endif
57  #if RH_ACT_9
58          case 9: // Do something
59              return;
60  #endif
61  #if RH_ACT_A
62          case 0xA: // Do something
63              return;
64  #endif
65  #if RH_ACT_B
66          case 0xB:
67              // Do something
68              return;
69  #endif
70  #if RH_ACT_C
71          case 0xC: // Do something
72              return;
73  #endif
74  #if RH_ACT_D
75          case 0xD: // Do something
76              return;
77  #endif
78  #if RH_ACT_E
79          case 0xE: // Do something
80              return;
81  #endif
82  #if RH_ACT_F
83          case 0xF: // Do something
84              return;
85  #endif
86          default:
87              return;
88      }
89  }
```

C.16.2.2. ActGetDataPointer()

```
90  static P_ACT_DATA
91  ActGetDataPointer(
92      uint32_t            act
93  )
94  {
95
96  #define RETURN_ACT_POINTER(N)  if(0x##N == act) return &ACT_##N;
97
98      FOR_EACH_ACT(RETURN_ACT_POINTER)
99
100     return (P_ACT_DATA)NULL;
101 }
```

C.16.2.3. _plat__ACT_GetImplemented()

This function tests to see if an ACT is implemented. It is a belt and suspenders function because the TPM should not be calling to to manipulate an ACT that is not implemented. However, this could help the simulator code which doesn't necessarily know if an ACT is implemented or not.

```
102 LIB_EXPORT int
103 _plat__ACT_GetImplemented(
104     uint32_t            act
```

```
105 )
106 {
107     return (ActGetDataPointer(act) != NULL);
108 }
```

C.16.2.4. _plat__ACT_GetRemaining()

This function returns the remaining time. If an update is pending, *newValue* is returned. Otherwise, the current counter value is returned. Note that since the timers keep running, the returned value can get stale immediately. The actual count value will be no greater than the returned value.

```
109 LIB_EXPORT uint32_t
110 _plat__ACT_GetRemaining(
111     uint32_t            act             //IN: the ACT selector
112 )
113 {
114     P_ACT_DATA              actData = ActGetDataPointer(act);
115     uint32_t                remain;
116 //
117     if(actData == NULL)
118         return 0;
119     remain = actData->remaining;
120     if(actData->pending)
121         remain = actData->newValue;
122     return remain;
123 }
```

C.16.2.5. _plat__ACT_GetSignaled()

```
124 LIB_EXPORT int
125 _plat__ACT_GetSignaled(
126     uint32_t            act         //IN: number of ACT to check
127 )
128 {
129     P_ACT_DATA              actData = ActGetDataPointer(act);
130 //
131     if(actData == NULL)
132         return 0;
133     return (int )actData->signaled;
134 }
```

C.16.2.6. _plat__ACT_SetSignaled()

```
135 LIB_EXPORT void
136 _plat__ACT_SetSignaled(
137     uint32_t            act,
138     int                 on
139 )
140 {
141     ActSignal(ActGetDataPointer(act), on);
142 }
```

C.16.2.7. _plat__ACT_GetPending()

```
143 LIB_EXPORT int
144 _plat__ACT_GetPending(
145     uint32_t            act         //IN: number of ACT to check
146 )
147 {
148     P_ACT_DATA              actData = ActGetDataPointer(act);
```

```
149     //
150         if(actData == NULL)
151             return 0;
152         return (int )actData->pending;
153     }
```

C.16.2.8. _plat__ACT_UpdateCounter()

This function is used to write the *newValue* for the counter. If an update is pending, then no update occurs and the function returns FALSE. If *setSignaled* is TRUE, then the ACT signaled state is SET and if *newValue* is 0, nothing is posted.

```
154     LIB_EXPORT int
155     _plat__ACT_UpdateCounter(
156         uint32_t            act,        // IN: ACT to update
157         uint32_t            newValue    // IN: the value to post
158     )
159     {
160         P_ACT_DATA          actData = ActGetDataPointer(act);
161      //
162         if(actData == NULL)
163             // actData doesn't exist but pretend update is pending rather than indicate
164             // that a retry is necessary.
165             return TRUE;
166         // if an update is pending then return FALSE so that there will be a retry
167         if(actData->pending != 0)
168             return FALSE;
169         actData->newValue = newValue;
170         actData->pending = TRUE;
171
172         return TRUE;
173     }
```

C.16.2.9. _plat__ACT_EnableTicks()

This enables and disables the processing of the once-per-second ticks. This should be turned off (*enable* = FALSE) by _TPM_Init() and turned on (*enable* = TRUE) by TPM2_Startup() after all the initializations have completed.

```
174     LIB_EXPORT void
175     _plat__ACT_EnableTicks(
176         int             enable
177     )
178     {
179         actTicksAllowed = enable;
180     }
```

C.16.2.10. ActDecrement()

If *newValue* is non-zero it is copied to *remaining* and then *newValue* is set to zero. Then *remaining* is decremented by one if it is not already zero. If the value is decremented to zero, then the associated event is signaled. If setting *remaining* causes it to be greater than 1, then the signal associated with the ACT is turned off.

```
181     static void
182     ActDecrement(
183         P_ACT_DATA              actData
184     )
185     {
186         // Check to see if there is an update pending
```

```
187         if(actData->pending)
188         {
189             // If this update will cause the count to go from non-zero to zero, set
190             // the newValue to 1 so that it will timeout when decremented below.
191             if((actData->newValue == 0) && (actData->remaining != 0))
192                actData->newValue = 1;
193             actData->remaining = actData->newValue;
194 
195             // Update processed
196             actData->pending = 0;
197         }
198         // no update so countdown if the count is non-zero but not max
199         if((actData->remaining != 0) && (actData->remaining != UINT32_MAX))
200         {
201             // If this countdown causes the count to go to zero, then turn the signal for
202             // the ACT on.
203             if((actData->remaining -= 1) == 0)
204                 ActSignal(actData, TRUE);
205         }
206         // If the current value of the counter is non-zero, then the signal should be
207         // off.
208         if(actData->signaled && (actData->remaining > 0))
209                 ActSignal(actData, FALSE);
210     }
```

C.16.2.11._plat__ACT_Tick()

This processes the once-per-second clock tick from the hardware. This is set up for the simulator to use the control interface to send ticks to the TPM. These ticks do not have to be on a per second basis. They can be as slow or as fast as desired so that the simulation can be tested.

```
211     LIB_EXPORT void
212     _plat__ACT_Tick(
213         void
214     )
215     {
216         // Ticks processing is turned off at certain times just to make sure that nothing
217         // strange is happening before pointers and things are
218         if(actTicksAllowed)
219         {
220             // Handle the update for each counter.
221     #define DECREMENT_COUNT(N)    ActDecrement(&ACT_##N);
222 
223             FOR_EACH_ACT(DECREMENT_COUNT)
224         }
225     }
```

C.16.2.12.ActZero()

This function initializes a single ACT

```
226     static void
227     ActZero(
228         uint32_t        act,
229         P_ACT_DATA      actData
230     )
231     {
232         actData->remaining = 0;
233         actData->newValue = 0;
234         actData->pending = 0;
235         actData->number = (uint8_t)act;
236         ActSignal(actData, FALSE);
```

```
237 }
```

C.16.2.13._plat__ACT_Initialize()

This function initializes the ACT hardware and data structures

```
238 LIB_EXPORT int
239 _plat__ACT_Initialize(
240     void
241 )
242 {
243     actTicksAllowed = 0;
244 #define ZERO_ACT(N)  ActZero(0x##N, &ACT_##N);
245     FOR_EACH_ACT(ZERO_ACT)
246 
247     return TRUE;
248 }
```

C.17 PlatformClock.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_CLOCK_H_
2   #define _PLATFORM_CLOCK_H_
3   #ifdef _MSC_VER
4   #include <sys/types.h>
5   #include <sys/timeb.h>
6   #else
7   #include <sys/time.h>
8   #include <time.h>
9   #endif
```

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.

```
10  #define     CLOCK_NOMINAL           30000
```

A 1% change in rate is 300 counts

```
11  #define     CLOCK_ADJUST_COARSE     300
```

A 0.1% change in rate is 30 counts

```
12  #define     CLOCK_ADJUST_MEDIUM     30
```

A minimum change in rate is 1 count

```
13  #define     CLOCK_ADJUST_FINE       1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```
14  #define     CLOCK_ADJUST_LIMIT      5000
15  #endif // _PLATFORM_CLOCK_H_
```

Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

D.2 TpmTcpProtocol.h

D.2.1. Introduction

TPM commands are communicated as BYTE streams on a TCP connection. The TPM command protocol is enveloped with the interface protocol described in this file. The command is indicated by a UINT32 with one of the values below. Most commands take no parameters return no TPM errors. In these cases the TPM interface protocol acknowledges that command processing is 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.2.2. Typedefs and Defines

```
#ifndef     TCP_TPM_PROTOCOL_H
#define     TCP_TPM_PROTOCOL_H
```

D.2.3. 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_ACT_GET_SIGNALED        26
#define TPM_TEST_FAILURE_MODE       30
```

D.2.4. Enumerations and Structures

```
enum TpmEndPointInfo
{
    tpmPlatformAvailable = 0x01,
    tpmUsesTbs = 0x02,
    tpmInRawMode = 0x04,
    tpmSupportsPP = 0x08
};
#ifdef _MSC_VER
```

```
37  #   pragma warning(push, 3)
38  #endif
```

Existing RPC interface type definitions retained so that the implementation can be re-used

```
39  typedef struct in_buffer
40  {
41      unsigned long BufferSize;
42      unsigned char *Buffer;
43  } _IN_BUFFER;
44  typedef unsigned char *_OUTPUT_BUFFER;
45  typedef struct out_buffer
46  {
47      uint32_t          BufferSize;
48      _OUTPUT_BUFFER    Buffer;
49  } _OUT_BUFFER;
50  #ifdef _MSC_VER
51  #   pragma warning(pop)
52  #endif
53  #ifndef WIN32
54  typedef unsigned long         DWORD;
55  typedef void                 *LPVOID;
56  #endif
57  #endif
```

D.3 TcpServer.c

D.3.1. Description

This file contains the socket interface to a TPM simulator.

D.3.2. Includes, Locals, Defines and Function Prototypes

```
#include "TpmBuildSwitches.h"
#include <stdio.h>
#ifdef _MSC_VER
#   pragma warning(push, 3)
#   include <windows.h>
#   include <winsock.h>
#   pragma warning(pop)
typedef int socklen_t;
#elif defined(__unix__)
#   include <string.h>
#   include <unistd.h>
#   include <errno.h>
#   include <stdint.h>
#   include <netinet/in.h>
#   include <sys/socket.h>
#   include <pthread.h>
#   define ZeroMemory(ptr, sz) (memset((ptr), 0, (sz)))
#   define closesocket(x) close(x)
#   define INVALID_SOCKET (-1)
#   define SOCKET_ERROR   (-1)
#   define WSAGetLastError() (errno)
#   define INT_PTR intptr_t
    typedef int SOCKET;
#else
#   error "Unsupported platform."
#endif
#ifndef TRUE
#   define TRUE    1
#endif
#ifndef FALSE
#   define FALSE   0
#endif
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include "TpmTcpProtocol.h"
#include "Manufacture_fp.h"
#include "TpmProfile.h"
#include "Simulator_fp.h"
#include "Platform_fp.h"
typedef int         BOOL;
```

To access key cache control in TPM

```
void RsaKeyCacheControl(int state);
#ifndef __IGNORE_STATE__
static uint32_t ServerVersion = 1;

#define MAX_BUFFER 1048576
char InputBuffer[MAX_BUFFER];       //The input data buffer for the simulator.
char OutputBuffer[MAX_BUFFER];      //The output data buffer for the simulator.

struct
{
```

```
    uint32_t    largestCommandSize;
    uint32_t    largestCommand;
    uint32_t    largestResponseSize;
    uint32_t    largestResponse;
} CommandResponseSizes = {0};

#endif // __IGNORE_STATE___

//** Functions

//*** CreateSocket()
// This function creates a socket listening on 'PortNumber'.
static int
CreateSocket(
    int                 PortNumber,
    SOCKET              *listenSocket
    )
{
    struct              sockaddr_in MyAddress;
    int res;
//
    // Initialize Winsock
#ifdef _MSC_VER
    WSADATA             wsaData;
    res = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if(res != 0)
    {
        printf("WSAStartup failed with error: %d\n", res);
        return -1;
    }
#endif
    // create listening socket
    *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
    if(INVALID_SOCKET == *listenSocket)
    {
        printf("Cannot create server listen socket.  Error is 0x%x\n",
               WSAGetLastError());
        return -1;
    }
    // bind the listening socket to the specified port
    ZeroMemory(&MyAddress, sizeof(MyAddress));
    MyAddress.sin_port = htons((short)PortNumber);
    MyAddress.sin_family = AF_INET;

    res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
    if(res == SOCKET_ERROR)
    {
        printf("Bind error.  Error is 0x%x\n", WSAGetLastError());
        return -1;
    }
    // listen/wait for server connections
    res = listen(*listenSocket, 3);
    if(res == SOCKET_ERROR)
    {
        printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
        return -1;
    }
    return 0;
}
```

D.3.2.1. PlatformServer()

This function processes incoming platform requests.

```
111  BOOL
112  PlatformServer(
113      SOCKET            s
114      )
115  {
116      BOOL                    OK = TRUE;
117      uint32_t                Command;
118  //
119      for(;;)
120      {
121          OK = ReadBytes(s, (char*)&Command, 4);
122          // client disconnected (or other error).  We stop processing this client
123          // and return to our caller who can stop the server or listen for another
124          // connection.
125          if(!OK)
126              return TRUE;
127          Command = ntohl(Command);
128          switch(Command)
129          {
130              case TPM_SIGNAL_POWER_ON:
131                  _rpc__Signal_PowerOn(FALSE);
132                  break;
133              case TPM_SIGNAL_POWER_OFF:
134                  _rpc__Signal_PowerOff();
135                  break;
136              case TPM_SIGNAL_RESET:
137                  _rpc__Signal_PowerOn(TRUE);
138                  break;
139              case TPM_SIGNAL_RESTART:
140                  _rpc__Signal_Restart();
141                  break;
142              case TPM_SIGNAL_PHYS_PRES_ON:
143                  _rpc__Signal_PhysicalPresenceOn();
144                  break;
145              case TPM_SIGNAL_PHYS_PRES_OFF:
146                  _rpc__Signal_PhysicalPresenceOff();
147                  break;
148              case TPM_SIGNAL_CANCEL_ON:
149                  _rpc__Signal_CancelOn();
150                  break;
151              case TPM_SIGNAL_CANCEL_OFF:
152                  _rpc__Signal_CancelOff();
153                  break;
154              case TPM_SIGNAL_NV_ON:
155                  _rpc__Signal_NvOn();
156                  break;
157              case TPM_SIGNAL_NV_OFF:
158                  _rpc__Signal_NvOff();
159                  break;
160              case TPM_SIGNAL_KEY_CACHE_ON:
161                  _rpc__RsaKeyCacheControl(TRUE);
162                  break;
163              case TPM_SIGNAL_KEY_CACHE_OFF:
164                  _rpc__RsaKeyCacheControl(FALSE);
165                  break;
166              case TPM_SESSION_END:
167                  // Client signaled end-of-session
168                  TpmEndSimulation();
169                  return TRUE;
170              case TPM_STOP:
171                  // Client requested the simulator to exit
172                  return FALSE;
173              case TPM_TEST_FAILURE_MODE:
174                  _rpc__ForceFailureMode();
175                  break;
176              case TPM_GET_COMMAND_RESPONSE_SIZES:
```

```
                OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
                                   sizeof(CommandResponseSizes));
                memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
                if(!OK)
                    return TRUE;
                break;
            case TPM_ACT_GET_SIGNALED:
            {
                UINT32 actHandle;
                OK = ReadUINT32(s, &actHandle);
                WriteUINT32(s, _rpc__ACT_GetSignaled(actHandle));
                break;
            }
            default:
                printf("Unrecognized platform interface command %d\n",
                       (int)Command);
                WriteUINT32(s, 1);
                return TRUE;
        }
        WriteUINT32(s, 0);
    }
}
```

D.3.2.2. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```
DWORD WINAPI
PlatformSvcRoutine(
    LPVOID           port
    )
{
    int                 PortNumber = (int)(INT_PTR)port;
    SOCKET              listenSocket, serverSocket;
    struct              sockaddr_in HerAddress;
    int                 res;
    socklen_t           length;
    BOOL                continueServing;
//
    res = CreateSocket(PortNumber, &listenSocket);
    if(res != 0)
    {
        printf("Create platform service socket fail\n");
        return res;
    }
    // Loop accepting connections one-by-one until we are killed or asked to stop
    // Note the platform service is single-threaded so we don't listen for a new
    // connection until the prior connection drops.
    do
    {
        printf("Platform server listening on port %d\n", PortNumber);

        // blocking accept
        length = sizeof(HerAddress);
        serverSocket = accept(listenSocket,
                              (struct sockaddr*) &HerAddress,
                              &length);
        if(serverSocket == INVALID_SOCKET)
        {
            printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
            return (DWORD)-1;
        }
        printf("Client accepted\n");

```

```
236             // normal behavior on client disconnection is to wait for a new client
237             // to connect
238             continueServing = PlatformServer(serverSocket);
239             closesocket(serverSocket);
240         } while(continueServing);
241 
242         return 0;
243     }
```

D.3.2.3. 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.

```
244     int
245     PlatformSignalService(
246         int             PortNumber
247         )
248     {
249     #if defined(_MSC_VER)
250         HANDLE              hPlatformSvc;
251         int                 ThreadId;
252         int                 port = PortNumber;
253     //
254         // Create service thread for platform signals
255         hPlatformSvc = CreateThread(NULL, 0,
256                                     (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
257                                     (LPVOID)(INT_PTR)port, 0, (LPDWORD)&ThreadId);
258         if(hPlatformSvc == NULL)
259         {
260             printf("Thread Creation failed\n");
261             return -1;
262         }
263         return 0;
264     #else
265         pthread_t           thread_id;
266         int                 ret;
267         int                 port = PortNumber;
268 
269         ret = pthread_create(&thread_id, NULL, (void*)PlatformSvcRoutine,
270                              (LPVOID)(INT_PTR)port);
271         if (ret == -1)
272         {
273             printf("pthread_create failed: %s", strerror(ret));
274         }
275         return ret;
276     #endif // _MSC_VER
277     }
```

D.3.2.4. RegularCommandService()

This function services regular commands.

```
278     int
279     RegularCommandService(
280         int             PortNumber
281         )
282     {
283         SOCKET              listenSocket;
284         SOCKET              serverSocket;
285         struct              sockaddr_in HerAddress;
286         int                 res;
287         socklen_t           length;
```

```
288         BOOL                    continueServing;
289     //
290         res = CreateSocket(PortNumber, &listenSocket);
291         if(res != 0)
292         {
293             printf("Create platform service socket fail\n");
294             return res;
295         }
296         // Loop accepting connections one-by-one until we are killed or asked to stop
297         // Note the TPM command service is single-threaded so we don't listen for
298         // a new connection until the prior connection drops.
299         do
300         {
301             printf("TPM command server listening on port %d\n", PortNumber);
302
303             // blocking accept
304             length = sizeof(HerAddress);
305             serverSocket = accept(listenSocket,
306                                   (struct sockaddr*) &HerAddress,
307                                   &length);
308             if(serverSocket == INVALID_SOCKET)
309             {
310                 printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
311                 return -1;
312             }
313             printf("Client accepted\n");
314
315             // normal behavior on client disconnection is to wait for a new client
316             // to connect
317             continueServing = TpmServer(serverSocket);
318             closesocket(serverSocket);
319         } while(continueServing);
320         return 0;
321     }
322     #if RH_ACT_0
```

D.3.2.5. SimulatorTimeServiceRoutine()

This function is called to service the time *ticks*.

```
323     static DWORD WINAPI
324     SimulatorTimeServiceRoutine(
325         LPVOID          notUsed
326     )
327     {
328         // All time is in ms
329         const INT64   tick = 1000;
330         UINT64  prevTime = _plat__RealTime();
331         INT64   timeout = tick;
332
333         (void)notUsed;
334
335         while (TRUE)
336         {
337             UINT64  curTime;
338
339     #if defined(_MSC_VER)
340             Sleep((DWORD)timeout);
341     #else
342             struct timespec req = {timeout / 1000, (timeout % 1000) * 1000}
343                             rem;
344             nanosleep(&req, &rem);
345     #endif // _MSC_VER
346             curTime = _plat__RealTime();
```

```
347
348             // May need to issue several ticks if the Sleep() took longer than asked,
349             // or no ticks at all, it Sleep() was interrupted prematurely.
350             while (prevTime < curTime - tick / 2)
351             {
352                 //printf("%05lld | %05lld\n",
353                 //       prevTime % 100000, (curTime - tick / 2) % 100000);
354                 _plat__ACT_Tick();
355                 prevTime += (UINT64)tick;
356             }
357             // Adjust the next timeout to keep the average interval of one second
358             timeout = tick + (prevTime - curTime);
359             //prevTime = curTime;
360             //printf("%04lld | c:%05lld | p:%05llu\n",
361             //           timeout, curTime % 100000, prevTime);
362         }
363         return 0;
364     }
```

D.3.2.6. ActTimeService()

This function starts a new thread waiting to wait for time ticks.

Return Value	Meaning
==0	success
!=0	failure

```
365     static int
366     ActTimeService(
367         void
368     )
369     {
370         static BOOL          running = FALSE;
371         int                  ret = 0;
372         if(!running)
373         {
374     #if defined(_MSC_VER)
375             HANDLE               hThr;
376             int                  ThreadId;
377         //
378             printf("Starting ACT thread...\n");
379             //  Don't allow ticks to be processed before TPM is manufactured.
380             _plat__ACT_EnableTicks(FALSE);
381
382             // Create service thread for ACT internal timer
383             hThr = CreateThread(NULL, 0,
384                 (LPTHREAD_START_ROUTINE)SimulatorTimeServiceRoutine,
385                                 (LPVOID)(INT_PTR)NULL, 0, (LPDWORD)&ThreadId);
386             if(hThr != NULL)
387                 CloseHandle(hThr);
388             else
389                 ret = -1;
390     #else
391             pthread_t            thread_id;
392     //
393             ret = pthread_create(&thread_id, NULL, (void*)PlatformSvcRoutine,
394                 (LPVOID)(INT_PTR)NULL);
395     #endif // _MSC_VER
396
397             if(ret != 0)
398                 printf("ACT thread Creation failed\n");
399             else
```

```
400             running = TRUE;
401     }
402     return ret;
403 }
404 #endif // RH_ACT_0
```

D.3.2.7. 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.

```
405 int
406 StartTcpServer(
407     int             PortNumber
408     )
409 {
410     int                 res;
411 //
412 #if RH_ACT_0 || 1
413     // Start the Time Service routine
414     res = ActTimeService();
415     if(res != 0)
416     {
417         printf("TimeService failed\n");
418         return res;
419     }
420 #endif
421 
422     // Start Platform Signal Processing Service
423     res = PlatformSignalService(PortNumber + 1);
424     if(res != 0)
425     {
426         printf("PlatformSignalService failed\n");
427         return res;
428     }
429     // Start Regular/DRTM TPM command service
430     res = RegularCommandService(PortNumber);
431     if(res != 0)
432     {
433         printf("RegularCommandService failed\n");
434         return res;
435     }
436     return 0;
437 }
```

D.3.2.8. ReadBytes()

This function reads the indicated number of bytes (*NumBytes*) into buffer from the indicated socket.

```
438 BOOL
439 ReadBytes(
440     SOCKET          s,
441     char           *buffer,
442     int             NumBytes
443     )
444 {
445     int                 res;
446     int                 numGot = 0;
447 //
448     while(numGot < NumBytes)
449     {
450         res = recv(s, buffer + numGot, NumBytes - numGot, 0);
```

```
451         if(res == -1)
452         {
453             printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
454             return FALSE;
455         }
456         if(res == 0)
457         {
458             return FALSE;
459         }
460         numGot += res;
461     }
462     return TRUE;
463 }
```

D.3.2.9. WriteBytes()

This function will send the indicated number of bytes (*NumBytes*) to the indicated socket

```
464 BOOL
465 WriteBytes(
466     SOCKET           s,
467     char            *buffer,
468     int              NumBytes
469     )
470 {
471     int                 res;
472     int                 numSent = 0;
473 //
474     while(numSent < NumBytes)
475     {
476         res = send(s, buffer + numSent, NumBytes - numSent, 0);
477         if(res == -1)
478         {
479             if(WSAGetLastError() == 0x2745)
480             {
481                 printf("Client disconnected\n");
482             }
483             else
484             {
485                 printf("Send error.  Error is 0x%x\n", WSAGetLastError());
486             }
487             return FALSE;
488         }
489         numSent += res;
490     }
491     return TRUE;
492 }
```

D.3.2.10. WriteUINT32()

Send 4 byte integer

```
493 BOOL
494 WriteUINT32(
495     SOCKET           s,
496     uint32_t         val
497     )
498 {
499     UINT32 netVal = htonl(val);
500 //
501     return WriteBytes(s, (char*)&netVal, 4);
502 }
```

D.3.2.11. ReadUINT32()

Function to read 4 byte integer from socket.

```
503 BOOL
504 ReadUINT32(
505     SOCKET           s,
506     UINT32          *val
507 )
508 {
509     UINT32 netVal;
510 //
511     if (!ReadBytes(s, (char*)&netVal, 4))
512         return FALSE;
513     *val = ntohl(netVal);
514     return TRUE;
515 }
```

D.3.2.12. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
516 BOOL
517 ReadVarBytes(
518     SOCKET           s,
519     char            *buffer,
520     uint32_t        *BytesReceived,
521     int              MaxLen
522     )
523 {
524     int                  length;
525     BOOL                 res;
526 //
527     res = ReadBytes(s, (char*)&length, 4);
528     if(!res) return res;
529     length = ntohl(length);
530     *BytesReceived = length;
531     if(length > MaxLen)
532     {
533         printf("Buffer too big.  Client says %d\n", length);
534         return FALSE;
535     }
536     if(length == 0) return TRUE;
537     res = ReadBytes(s, buffer, length);
538     if(!res) return res;
539     return TRUE;
540 }
```

D.3.2.13. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
541 BOOL
542 WriteVarBytes(
543     SOCKET           s,
544     char            *buffer,
545     int              BytesToSend
546     )
547 {
548     uint32_t             netLength = htonl(BytesToSend);
```

```
549         BOOL res;
550     //
551         res = WriteBytes(s, (char*)&netLength, 4);
552         if(!res)
553             return res;
554         res = WriteBytes(s, buffer, BytesToSend);
555         if(!res)
556             return res;
557         return TRUE;
558     }
```

D.3.2.14. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
559     BOOL
560     TpmServer(
561         SOCKET           s
562         )
563     {
564         uint32_t             length;
565         uint32_t             Command;
566         BYTE                 locality;
567         BOOL                 OK;
568         int                  result;
569         int                  clientVersion;
570         _IN_BUFFER           InBuffer;
571         _OUT_BUFFER          OutBuffer;
572     //
573         for(;;)
574         {
575             OK = ReadBytes(s, (char*)&Command, 4);
576             // client disconnected (or other error).  We stop processing this client
577             // and return to our caller who can stop the server or listen for another
578             // connection.
579             if(!OK)
580                 return TRUE;
581             Command = ntohl(Command);
582             switch(Command)
583             {
584                 case TPM_SIGNAL_HASH_START:
585                     _rpc__Signal_Hash_Start();
586                     break;
587                 case TPM_SIGNAL_HASH_END:
588                     _rpc__Signal_HashEnd();
589                     break;
590                 case TPM_SIGNAL_HASH_DATA:
591                     OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
592                     if(!OK) return TRUE;
593                     InBuffer.Buffer = (BYTE*)InputBuffer;
594                     InBuffer.BufferSize = length;
595                     _rpc__Signal_Hash_Data(InBuffer);
596                     break;
597                 case TPM_SEND_COMMAND:
598                     OK = ReadBytes(s, (char*)&locality, 1);
599                     if(!OK)
600                         return TRUE;
601                     OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
602                     if(!OK)
603                         return TRUE;
604                     InBuffer.Buffer = (BYTE*)InputBuffer;
605                     InBuffer.BufferSize = length;
606                     OutBuffer.BufferSize = MAX_BUFFER;
607                     OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;
```

```
608                 // record the number of bytes in the command if it is the largest
609                 // we have seen so far.
610                 if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
611                 {
612                     CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
613                     memcpy(&CommandResponseSizes.largestCommand,
614                            &InputBuffer[6], sizeof(UINT32));
615                 }
616                 _rpc__Send_Command(locality, InBuffer, &OutBuffer);
617                 // record the number of bytes in the response if it is the largest
618                 // we have seen so far.
619                 if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
620                 {
621                     CommandResponseSizes.largestResponseSize
622                         = OutBuffer.BufferSize;
623                     memcpy(&CommandResponseSizes.largestResponse,
624                            &OutputBuffer[6], sizeof(UINT32));
625                 }
626                 OK = WriteVarBytes(s,
627                                    (char*)OutBuffer.Buffer,
628                                    OutBuffer.BufferSize);
629                 if(!OK)
630                     return TRUE;
631                 break;
632             case TPM_REMOTE_HANDSHAKE:
633                 OK = ReadBytes(s, (char*)&clientVersion, 4);
634                 if(!OK)
635                     return TRUE;
636                 if(clientVersion == 0)
637                 {
638                     printf("Unsupported client version (0).\n");
639                     return TRUE;
640                 }
641                 OK &= WriteUINT32(s, ServerVersion);
642                 OK &= WriteUINT32(s, tpmInRawMode
643                                   | tpmPlatformAvailable | tpmSupportsPP);
644                 break;
645             case TPM_SET_ALTERNATIVE_RESULT:
646                 OK = ReadBytes(s, (char*)&result, 4);
647                 if(!OK)
648                     return TRUE;
649                 // Alternative result is not applicable to the simulator.
650                 break;
651             case TPM_SESSION_END:
652                 // Client signaled end-of-session
653                 return TRUE;
654             case TPM_STOP:
655                 // Client requested the simulator to exit
656                 return FALSE;
657             default:
658                 printf("Unrecognized TPM interface command %d\n", (int)Command);
659                 return TRUE;
660         }
661         OK = WriteUINT32(s, 0);
662         if(!OK)
663             return TRUE;
664     }
665 }
```

D.4 TPMCmdp.c

D.4.1. Description

This file contains the functions that process the commands received on the control port or the command port of the simulator. The control port is used to allow simulation of hardware events (such as, _TPM_Hash_Start()) to test the simulated TPM's reaction to those events. This improves code coverage of the testing.

D.4.2. Includes and Data Definitions

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
#include "TpmBuildSwitches.h"
#ifdef _MSC_VER
#   pragma warning(push, 3)
#   include <windows.h>
#   include <winsock.h>
#   pragma warning(pop)
#elif defined(__unix__)
    typedef int SOCKET;
#else
#   error "Unsupported platform."
#endif
#ifndef TRUE
#   define TRUE    1
#endif
#ifndef FALSE
#   define FALSE   0
#endif
#include "Platform_fp.h"
#include "ExecCommand_fp.h"
#include "Manufacture_fp.h"
#include "_TPM_Init_fp.h"
#include "_TPM_Hash_Start_fp.h"
#include "_TPM_Hash_Data_fp.h"
#include "_TPM_Hash_End_fp.h"
#include "TpmFail_fp.h"
#include "TpmTcpProtocol.h"
#include "Simulator_fp.h"
static BOOL     s_isPowerOn = FALSE;

//** Functions

//*** Signal_PowerOn()
// This function processes a power-on indication. Among other things, it
// calls the _TPM_Init() handler.
void
_rpc__Signal_PowerOn(
    BOOL        isReset
    )
{
    // if power is on and this is not a call to do TPM reset then return
    if(s_isPowerOn && !isReset)
        return;
    // If this is a reset but power is not on, then return
    if(isReset && !s_isPowerOn)
        return;
    // Unless this is just a reset, pass power on signal to platform
    if(!isReset)
        _plat__Signal_PowerOn();
```

```
52      // Power on and reset both lead to _TPM_Init()
53      _plat__Signal_Reset();
54
55      // Set state as power on
56      s_isPowerOn = TRUE;
57  }
```

D.4.2.1. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```
58  void
59  _rpc__Signal_Restart(
60      void
61      )
62  {
63      _plat__TimerRestart();
64  }
```

D.4.2.2. 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      )
69  {
70      if(s_isPowerOn)
71          // Pass power off signal to platform
72          _plat__Signal_PowerOff();
73      // This could be redundant, but...
74      s_isPowerOn = FALSE;
75
76      return;
77  }
```

D.4.2.3. _rpc__ForceFailureMode()

This function is used to debug the Failure Mode logic of the TPM. It will set a flag in the TPM code such that the next call to TPM2_SelfTest() will result in a failure, putting the TPM into Failure Mode.

```
78  void
79  _rpc__ForceFailureMode(
80      void
81      )
82  {
83      SetForceFailureMode();
84      return;
85  }
```

D.4.2.4. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```
86  void
87  _rpc__Signal_PhysicalPresenceOn(
88      void
```

```
89      )
90  {
91      // If TPM power is on
92      if(s_isPowerOn)
93          // Pass physical presence on to platform
94          _plat__Signal_PhysicalPresenceOn();
95      return;
96  }
```

D.4.2.5. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```
97  void
98  _rpc__Signal_PhysicalPresenceOff(
99      void
100     )
101 {
102     // If TPM is power on
103     if(s_isPowerOn)
104         // Pass physical presence off to platform
105         _plat__Signal_PhysicalPresenceOff();
106     return;
107 }
```

D.4.2.6. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```
108 void
109 _rpc__Signal_Hash_Start(
110     void
111     )
112 {
113     // If TPM power is on
114     if(s_isPowerOn)
115         // Pass _TPM_Hash_Start signal to TPM
116         _TPM_Hash_Start();
117     return;
118 }
```

D.4.2.7. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```
119 void
120 _rpc__Signal_Hash_Data(
121     _IN_BUFFER       input
122     )
123 {
124     // If TPM power is on
125     if(s_isPowerOn)
126         // Pass _TPM_Hash_Data signal to TPM
127         _TPM_Hash_Data(input.BufferSize, input.Buffer);
128     return;
129 }
```

D.4.2.8. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```
130 void
131 _rpc__Signal_HashEnd(
132     void
133     )
134 {
135     // If TPM power is on
136     if(s_isPowerOn)
137         // Pass _TPM_HashEnd signal to TPM
138         _TPM_Hash_End();
139     return;
140 }
```

D.4.2.9. _rpc__Send_Command()

This is the interface to the TPM code.

```
141 void
142 _rpc__Send_Command(
143     unsigned char    locality,
144     _IN_BUFFER       request,
145     _OUT_BUFFER     *response
146     )
147 {
148     // If TPM is power off, reject any commands.
149     if(!s_isPowerOn)
150     {
151         response->BufferSize = 0;
152         return;
153     }
154     // Set the locality of the command so that it doesn't change during the command
155     _plat__LocalitySet(locality);
156     // Do implementation-specific command dispatch
157     _plat__RunCommand(request.BufferSize, request.Buffer,
158                       &response->BufferSize, &response->Buffer);
159     return;
160 }
```

D.4.2.10. _rpc__Signal_CancelOn()

This function is used to turn on the indication to cancel a command in process. An executing command is not interrupted. The command code may periodically check this indication to see if it should abort the current command processing and returned TPM_RC_CANCELLED.

```
161 void
162 _rpc__Signal_CancelOn(
163     void
164     )
165 {
166     // If TPM power is on
167     if(s_isPowerOn)
168         // Set the platform canceling flag.
169         _plat__SetCancel();
170     return;
171 }
```

D.4.2.11. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.

```
172 void
173 _rpc__Signal_CancelOff(
```

```
174         void
175         )
176     {
177         // If TPM power is on
178         if(s_isPowerOn)
179             // Set the platform canceling flag.
180             _plat__ClearCancel();
181         return;
182     }
```

D.4.2.12. _rpc__Signal_NvOn()

In a system where the NV memory used by the TPM is not within the TPM, the NV may not always be available. This function turns on the indicator that indicates that NV is available.

```
183     void
184     _rpc__Signal_NvOn(
185         void
186         )
187     {
188         // If TPM power is on
189         if(s_isPowerOn)
190             // Make the NV available
191             _plat__SetNvAvail();
192         return;
193     }
```

D.4.2.13. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```
194     void
195     _rpc__Signal_NvOff(
196         void
197         )
198     {
199         // If TPM power is on
200         if(s_isPowerOn)
201             // Make NV not available
202             _plat__ClearNvAvail();
203         return;
204     }
205     void RsaKeyCacheControl(int state);
```

D.4.2.14. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulation.

```
206     void
207     _rpc__RsaKeyCacheControl(
208         int                 state
209         )
210     {
211     #if USE_RSA_KEY_CACHE
212         RsaKeyCacheControl(state);
213     #else
214         NOT_REFERENCED(state);
215     #endif
216         return;
217     }
218     #define TPM_RH_ACT_0        0x40000110
```

D.4.2.15. _rpc__ACT_GetSignaled()

This function is used to count the ACT second tick.

```
219  BOOL
220  _rpc__ACT_GetSignaled(
221      UINT32 actHandle
222  )
223  {
224      // If TPM power is on
225      if (s_isPowerOn)
226          // Query the platform
227          return _plat__ACT_GetSignaled(actHandle - TPM_RH_ACT_0);
228      return FALSE;
229  }
```

D.5 TPMCmds.c

D.5.1. Description

This file contains the entry point for the simulator.

D.5.2. Includes, Defines, Data Definitions, and Function Prototypes

```
#include "TpmBuildSwitches.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <ctype.h>
#include <string.h>
#ifdef _MSC_VER
#   pragma warning(push, 3)
#   include <windows.h>
#   include <winsock.h>
#   pragma warning(pop)
#elif defined(__unix__)
#   define _strcmpi strcasecmp
    typedef int SOCKET;
#else
#   error "Unsupported platform."
#endif
#ifndef TRUE
#   define TRUE    1
#endif
#ifndef FALSE
#   define FALSE   0
#endif
#include "TpmTcpProtocol.h"
#include "Manufacture_fp.h"
#include "Platform_fp.h"
#include "Simulator_fp.h"
#define PURPOSE \
"TPM 2.0 Reference Simulator.\n"  \
"Copyright (c) Microsoft Corporation. All rights reserved."
#define DEFAULT_TPM_PORT 2321
```

Information about command line arguments (does not include program name)

```
static uint32_t     s_ArgsMask = 0;     // Bit mask of unmatched command line args
static int          s_Argc = 0;
static const char **s_Argv = NULL;

//** Functions

#if DEBUG
//*** Assert()
// This function implements a run-time assertion.
// Computation of its parameters must not result in any side effects, as these
// computations will be stripped from the release builds.
static void Assert (BOOL cond, const char* msg)
{
    if (cond)
        return;
    fputs(msg, stderr);
    exit(2);
}
#else
#define Assert(cond, msg)
```

```
#endif
```

D.5.2.1. Usage()

This function prints the proper calling sequence for the simulator.

```
static void
Usage(
    const char          *programName
)
{
    fprintf(stderr, "%s\n\n", PURPOSE);
    fprintf(stderr, "Usage:  %s [PortNum] [opts]\n\n"
        "Starts the TPM server listening on TCP port PortNum (by default %d).\n\n"
        "An option can be in the short form (one letter preceded with '-' or '/')\n"
        "or in the full form (preceded with '--' or no option marker at all).\n"
        "Possible options are:\n"
        "   -h (--help) or ? - print this message\n"
        "   -m (--manufacture) - forces NV state of the TPM simulator to be "
        "(re)manufactured\n",
        programName, DEFAULT_TPM_PORT);
    exit(1);
}
```

D.5.2.2. CmdLineParser_Init()

This function initializes command line option parser.

```
static BOOL
CmdLineParser_Init(
    int argc,
    char *argv[],
    int maxOpts
    )
{
    if (argc == 1)
        return FALSE;

    if (maxOpts && (argc - 1) > maxOpts)
    {
        fprintf(stderr, "No more than %d options can be specified\n\n", maxOpts);
        Usage(argv[0]);
    }

    s_Argc = argc - 1;
    s_Argv = (const char**)(argv + 1);
    s_ArgsMask = (1 << s_Argc) - 1;
    return TRUE;
}
```

D.5.2.3. CmdLineParser_More()

Returns true if there are unparsed options still.

```
static BOOL
CmdLineParser_More(
    void
)
{
    return s_ArgsMask != 0;
}
```

D.5.2.4. CmdLineParser_IsOpt()

This function determines if the given command line parameter represents a valid option.

```
 98  static BOOL
 99  CmdLineParser_IsOpt(
100      const char* opt,        // Command line parameter to check
101      const char* optFull,    // Expected full name
102      const char* optShort,   // Expected short (single letter) name
103      BOOL dashed             // The parameter is preceded by a single dash
104      )
105  {
106      return 0 == strcmp(opt, optFull)
107          || (optShort && opt[0] == optShort[0] && opt[1] == 0)
108          || (dashed && opt[0] == '-' && 0 == strcmp(opt + 1, optFull));
109  }
```

D.5.2.5. CmdLineParser_IsOptPresent()

This function determines if the given command line parameter represents a valid option.

```
110  static BOOL
111  CmdLineParser_IsOptPresent(
112      const char* optFull,
113      const char* optShort
114      )
115  {
116      int         i;
117      int         curArgBit;
118      Assert(s_Argv != NULL,
119          "InitCmdLineOptParser(argc, argv) has not been invoked\n");
120      Assert(optFull && optFull[0],
121          "Full form of a command line option must be present.\n"
122          "If only a short (single letter) form is supported, it must be"
123          "specified as the full one.\n");
124      Assert(!optShort || (optShort[0] && !optShort[1]),
125          "If a short form of an option is specified, it must consist "
126          "of a single letter only.\n");
127
128      if (!CmdLineParser_More())
129          return FALSE;
130
131      for (i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <<= 1)
132      {
133          const char* opt = s_Argv[i];
134          if (   (s_ArgsMask & curArgBit) && opt
135              && (   0 == strcmp(opt, optFull)
136                  || (   (opt[0] == '/' || opt[0] == '-')
137                      && CmdLineParser_IsOpt(opt + 1, optFull, optShort,
138                                             opt[0] == '-'))))
139          {
140              s_ArgsMask ^= curArgBit;
141              return TRUE;
142          }
143      }
144      return FALSE;
145  }
```

D.5.2.6. CmdLineParser_IsOptPresent()

This function notifies the parser that no more options are needed.

```
146 static void
147 CmdLineParser_Done(
148     const char          *programName
149 )
150 {
151     char delim = ':';
152     int         i;
153     int         curArgBit;
154 
155     if (!CmdLineParser_More())
156         return;
157 
158     fprintf(stderr, "Command line contains unknown option%s",
159             s_ArgsMask & (s_ArgsMask - 1) ? "s" : "");
160     for (i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <<= 1)
161     {
162         if (s_ArgsMask & curArgBit)
163         {
164             fprintf(stderr, "%c %s", delim, s_Argv[i]);
165             delim = ',';
166         }
167     }
168     fprintf(stderr, "\n\n");
169     Usage(programName);
170 }
```

D.5.2.7. main()

This is the main entry point for the simulator. It registers the interface and starts listening for clients

```
171 int
172 main(
173     int              argc,
174     char            *argv[]
175     )
176 {
177     BOOL    manufacture = FALSE;
178     int     PortNum = DEFAULT_TPM_PORT;
179 
180     // Parse command line options
181 
182     if (CmdLineParser_Init(argc, argv, 2))
183     {
184         if (   CmdLineParser_IsOptPresent("?", "?")
185             || CmdLineParser_IsOptPresent("help", "h"))
186         {
187             Usage(argv[0]);
188         }
189         if (CmdLineParser_IsOptPresent("manufacture", "m"))
190         {
191             manufacture = TRUE;
192         }
193         if (CmdLineParser_More())
194         {
195             int     i;
196             for (i = 0; i < s_Argc; ++i)
197             {
198                 char *nptr = NULL;
199                 int portNum = (int)strtol(s_Argv[i], &nptr, 0);
200                 if (s_Argv[i] != nptr)
201                 {
202                     // A numeric option is found
203                     if(!*nptr && portNum > 0 && portNum < 65535)
204                     {
```

```
205                         PortNum = portNum;
206                         s_ArgsMask ^= 1 << i;
207                         break;
208                     }
209                     fprintf(stderr, "Invalid numeric option %s\n\n", s_Argv[i]);
210                     Usage(argv[0]);
211                 }
212             }
213         }
214         CmdLineParser_Done(argv[0]);
215     }
216     printf("LIBRARY_COMPATIBILITY_CHECK is %s\n",
217         (LIBRARY_COMPATIBILITY_CHECK ? "ON" : "OFF"));
218     // Enable NV memory
219     _plat__NVEnable(NULL);
220 
221     if (manufacture || _plat__NVNeedsManufacture())
222     {
223         printf("Manufacturing NV state...\n");
224         if(TPM_Manufacture(1) != 0)
225         {
226             // if the manufacture didn't work, then make sure that the NV file doesn't
227             // survive. This prevents manufacturing failures from being ignored the
228             // next time the code is run.
229             _plat__NVDisable(1);
230             exit(1);
231         }
232         // Coverage test - repeated manufacturing attempt
233         if(TPM_Manufacture(0) != 1)
234         {
235             exit(2);
236         }
237         // Coverage test - re-manufacturing
238         TPM_TearDown();
239         if(TPM_Manufacture(1) != 0)
240         {
241             exit(3);
242         }
243     }
244     // Disable NV memory
245     _plat__NVDisable(0);
246 
247     StartTcpServer(PortNum);
248     return EXIT_SUCCESS;
249 }
```