Trusted Platform Module Library

Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.38

September 29, 2016

Contact: admin@trustedcomputinggroup.org

TCG Published

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

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

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

CONTENTS

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

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

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

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

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

2 Terms and definitions

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

3 Symbols and abbreviated terms

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

4 Automation

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

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

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

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

4.1 Configuration Parser

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

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

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

4.2 Structure Parser

4.2.1 Introduction

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

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

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

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

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

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

4.2.2 Unmarshaling Code Prototype

4.2.2.1 Simple Types and Structures

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

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

Where:

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

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

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

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

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

4.2.2.2 Union Types

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

The function prototype for a union has the form:

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

where:

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

4.2.2.3 Null Types

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

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

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

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

4.2.2.4 Arrays

Any data type may be included in an array. The function prototype use to unmarshal an array for a **TYPE** is

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

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

4.2.3 Marshaling Code Function Prototypes

4.2.3.1 Simple Types and Structures

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

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

Where:

TYPE	name of the data type or structure
***source**	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
****buffer**	location in the output buffer where the first octet of the **TYPE** is to be placed
***size**	number of octets remaining in ****buffer**.

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

If **buffer** is not a NULL pointer, data is marshaled, ***buffer** is advanced to point to the first octet of the next location in the output buffer, and the called routine will return the number of octets marshaled into ****buffer**. This occurs even if **size** is a NULL pointer. If **size** is a not NULL pointer ***size** is reduced by the number of octets placed in the buffer.

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

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

4.2.3.2 Union Types

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

The function prototype for a union has the form:

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

The parameters have a similar meaning as those in 4.2.2.2 but the data movement is from **source** to **buffer**.

4.2.3.3 Arrays

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

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

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

4.3 Part 3 Parsing

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

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

Part 3 parsing also produces special function prototype files as described in 4.4**Error! Reference source not found.**.

4.4 Function Prototypes

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

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

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

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

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

```
[[create_fp_h]]
```

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

[[BaseTypes_h]]

5.3 Bits_fp.h

[[Bits_fp_h]]

5.4 Bool.h

[[Bool_h]]

5.5 Capabilities.h

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

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

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

[[Capabilities_h]]

5.6 CommandAttributeData.h

[[CommandAttributeData_h]]

5.7 CommandAttributes.h

[[CommandAttributes_h]]

5.8 CommandDispatchData.h

[[CommandDispatchData_h]]

5.9 Commands.h

[[Commands_h]]

5.10 CompilerDependencies.h

[[CompilerDependencies_h]]

5.11 Global.h

[[Global_h]]

5.12 GpMacros.h

[[GpMacros_h]]

5.13 InternalRoutines.h

[[InternalRoutines_h]]

5.14 LibSupport.h

[[LibSupport_h]]

5.15 NV.h

[[NV_h]]

5.16 PRNG_TestVectors.h

[[PRNG_TestVectors_h]]

5.17 SelfTest.h

[[SelfTest_h]]

5.18 SupportLibraryFunctionPrototypes_fp.h

[[SupportLibraryFunctionPrototypes_fp_h]]

5.19 TPMB.h

[[TPMB_h]]

5.20 Tpm.h

[[Tpm_h]]

5.21 TpmBuildSwitches.h

[[TpmBuildSwitches_h]]

5.22 TpmError.h

[[TpmError_h]]

5.23 TpmTypes.h

[[TpmTypes_h]]

5.24 VendorStrng.h

[[VendorString_h]]

5.25 swap.h

[[swap_h]]

6 Main

6.1 Introduction

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

6.2 ExecCommand.c

[[ExecCommand]]

6.3 CommandDispatcher.c

6.3.1 Introduction

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;

NOTE Unlike other unmarshaling functions, *parmBufferStart* does not advance. *parmBufferSize* Is reduced.

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

NOTE 1 The output buffer is the return from the MemoryGetResponseBuffer() function. It includes the header, handles, response parameters, and authorization area. *respParmSize* is the response parameter size, and does not include the header, handles, or authorization area.

NOTE 2 The reference implementation is permitted to do compare operations over a union as a byte array. Therefore, the command parameter *in* structure must be initialized (e.g., zeroed) before unmarshaling so that the compare operation is valid in cases where some bytes are unused.

[[CommandDispatcher]]

6.4 SessionProcess.c

[[SessionProcess]]

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)

[[Attest_spt]]

7.3 Context Management Command Support (Context_spt.c)

[[Context_spt]]

7.4 Policy Command Support (Policy_spt.c)

[[Policy_spt]]

7.5 NV Command Support (NV_spt.c)

[[NV_spt]]

7.6 Object Command Support (Object_spt.c)

[[Object_spt]]

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

[[EncryptDecrypt_spt]]

8 Subsystem

8.1 CommandAudit.c

[[CommandAudit]]

8.2 DA.c

[[DA]]

8.3 Hierarchy.c

[[Hierarchy]]

8.4 NVDynamic.c

[[NVDynamic]]

8.5 NVReserved.c

[[NVReserved]]

8.6 Object.c

[[Object]]

8.7 PCR.c

[[PCR]]

8.8 PP.c

[[PP]]

8.9 Session.c

[[Session]]

8.10 Time.c

[[Time]]

9 Support

9.1 AlgorithmCap.c

[[AlgorithmCap]]

9.2 Bits.c

[[Bits]]

9.3 CommandCodeAttributes.c

[[CommandCodeAttributes]]

9.4 Entity.c

[[Entity]]

9.5 Global.c

[[Global]]

9.6 Handle.c

[[Handle]]

9.7 IoBuffers.c

[[IoBuffers]]

9.8 Locality.c

[[Locality]]

9.9 Manufacture.c

[[Manufacture]]

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
```

```
TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

An additional script is used to do the work that might be done by a linker or globally optimizing compiler. It searches for functions like TPMI_DH_OBJECT_Marshal() that do nothing but call another function and replaces the function with a #define.

```
#define TPMI_DH_OBJECT_Marshal(source, buffer, size)    \
      UINT32_Marshal((UINT32 *)source, buffer, size)
```

When replacing the function with a #define, the #define is placed in marshal_fp.h and the function body is removed from marshal.c.

9.10.3 Unmarshal and Marshal a Union

In TPM 2.0 Part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTE The Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

From this table, the following unmarshaling code is generated.

```
TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
                            UINT32 selector)
{
    switch(selector) {
#ifdef TPM_ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal(
                        (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
#endif
#ifdef TPM_ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal(
                    (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
#endif
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal(
                                (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
#endif
#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal(
                                (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
#endif
```

```
26      }
27      return TPM_RC_SELECTOR;
28  }
```

NOTE The `#ifdef/#endif` directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
1   UINT16
2   TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                             UINT32 selector)
4   {
5       switch(selector) {
6   #ifdef TPM_ALG_KEYEDHASH
7           case TPM_ALG_KEYEDHASH:
8               return TPMS_KEYEDHASH_PARMS_Marshal(
9                           (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10  #endif
11  #ifdef TPM_ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Marshal(
14                          (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15  #endif
16  #ifdef TPM_ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Marshal(
19                           (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20  #endif
21  #ifdef TPM_ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Marshal(
24                           (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25  #endif
26      }
27      assert(1);
28      return 0;
29  }
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

9.10.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLIC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	"algorithm" associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
1   TPM_RC
2   TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
3   {
4       TPM_RC    result;
5       result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                       buffer, size, flag);
11      if(result != TPM_RC_SUCCESS)
12          return result;
13      result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                     buffer, size);
15      if(result != TPM_RC_SUCCESS)
16          return result;
17      result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                      buffer, size);
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                           buffer, size, (UINT32)target->type);
24      if(result != TPM_RC_SUCCESS)
25          return result;
26
27      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                        buffer, size, (UINT32)target->type);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31
32      return TPM_RC_SUCCESS;
33  }
```


The marshaling code for the TPMT_PUBLIC structure is:

```
UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
                                   (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
                                  (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
;
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                           (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));

    result = (UINT16)(result + TPM2B_DIGEST_Marshal(
                                (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));

    result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
                           (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
                                                                  (UINT32)source->type));

    result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
                                  (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
                                                                  (UINT32)source->type));

    return result;
}
```

9.10.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
TPM_RC
TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    if( (target->count < 2))    // This check is triggered by the {2:} notation
                                // on 'count'
        return TPM_RC_SIZE;

    if((target->count) > 8)     // This check is triggered by the {:8} notation
```

```
                                    // on 'digests'.
        return TPM_RC_SIZE;

    result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
                                          buffer, size, (INT32)(target->count));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}

```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
UINT16
TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
                                                                        size));
    result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
                                    (TPM2B_DIGEST *)(source->digests), buffer, size,
                                    (INT32)(source->count)));

    return result;
}
```

The marshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```


9.10.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.10.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

When marshaling a TPM2B where the second member is a BYTE array, the size parameter indicates the size of the array. The second member can also be a structure. In this case, the caller does not prefill the size member. The marshaling code must marshal the structure and then back fill the calculated size.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
1   TPM_RC
2   TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8       // if size equal to 0, the rest of the structure is a zero buffer
9       //  so stop processing
10      if(target->t.size == 0)
11          return TPM_RC_SUCCESS;
12      if((target->t.size) > 1024)    // This check is triggered by the {:1024}
13                                     // notation on 'buffer'
14          return TPM_RC_SIZE;
15      result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
16                                    (INT32)(target->t.size));
17      if(result != TPM_RC_SUCCESS)
18          return result;
19      return TPM_RC_SUCCESS;
20  }
```

using these structure definitions:

```
1   typedef union {
2       struct {
3           UINT16        size;
4           BYTE          buffer[1024];
5       }             t;
6       TPM2B         b;
7   } TPM2B_EVENT;
```


9.11 MathOnByteBuffers.c

[[MathOnByteBuffers]]

9.12 Memory.c

[[Memory]]

9.13 Power.c

[[Power]]

9.14 PropertyCap.c

[[PropertyCap]]

9.15 Response.c

[[Response]]

9.16 ResponseCodeProcessing.c

[[ResponseCodeProcessing]]

9.17 TpmFail.c

[[TpmFail]]

10 Cryptographic Functions

10.1 Headers

10.1.1 BnValues.h

[[BnValues_h]]

10.1.2 CryptEcc.h

[[CryptEcc_h]]

10.1.3 CryptHash.h

[[CryptHash_h]]

10.1.4 CryptHashData.h

[[CryptHashData_h]]

10.1.5 CryptRand.h

[[CryptRand_h]]

10.1.6 CryptRsa.h

[[CryptRsa_h]]

10.1.7 CryptTest.h

[[CryptTest_h]]

10.1.8 HashTestData.h

[[HashTestData_h]]

10.1.9 RsaTestData.h

[[RsaTestData_h]]

10.1.10 SymmetricTestData.h

[[SymmetricTestData_h]]

10.1.11 SymmetricTest.h

[[SymmetricTest_h]]

10.1.12 EccTestData.h

[[EccTestData_h]]

10.2 Source

10.2.1 AlgorithmTests.c

[[AlgorithmTests]]

10.2.2 BnConvert.c

[[BnConvert]]

10.2.3 BnEccData.c

[[BnEccData]]

10.2.4 BnMath.c

[[BnMath]]

10.2.5 BnMemory.c

[[BnMemory]]

10.2.6 CryptUtil.c

[[CryptUtil]]

10.2.7 CryptSelfTest.c

[[CryptSelfTest]]

10.2.8 CryptDataEcc.c

[[CryptDataEcc]]

10.2.9 CryptDes.c

[[CryptDes]]

10.2.10 CryptEccKeyExchange.c

[[CryptEccKeyExchange]]

10.2.11 CryptEccMain.c

[[CryptEccMain]]

10.2.12 CryptEccSignature.c

[[CryptEccSignature]]

10.2.13 CryptHash.c

[[CryptHash]]

10.2.14 CryptHashData.c

[[CryptHashData]]

10.2.15 CryptPrime.c

[[CryptPrime]]

10.2.16 CryptPrimeSieve.c

[[CryptPrimeSieve]]

10.2.17 CryptRand.c

[[CryptRand]]

10.2.18 CryptRsa.c

[[CryptRsa]]

10.2.19 CryptSym.c

[[CryptSym]]

10.2.20 PrimeData.c

[[PrimeData]]

10.2.21 RsaKeyCache.c

[[RsaKeyCache]]

10.2.22 Ticket.c

[[Ticket]]

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are used to define a TPM profile. Some of the values are derived from the values in the *TCG Algorithm Registry* [19] and others are simply chosen by the manufacturer. The values here are chosen to represent a full featured TPM so that all of the TPM's capabilities can be simulated and tested. This file would change based on the implementation.

NOTE The file listed below was generated by an automated tool using three documents as inputs. They are:

1) The TCG_Algorithm Registery,
2) Part 2 of this specification, and
3) A purpose-built document that contains vendor-specific information in tables.

A.2 Implementation.h

[[Implementation_h]]

Annex B
(informative)
Library-Specific

B.1 Introduction

This clause contains the files that are specific to a cryptographic library used by the TPM code.

Three categories are defined for cryptographic functions:

1) big number math (asymmetric cryptography),
2) symmetric ciphers, and
3) hash functions.

The code is structured to make it possible to use different libraries for different categories. For example, one might choose to use OpenSSL for its math library, but use a different library for hashing and symmetric cryptography. Since OpenSSL supports all three categories, it might be more typical to combine libraries of specific functions; that is, one library might only contain block ciphers while another supports big number math.

B.2 OpenSSL-Specific Files

B.2.1. Introduction

The following files are specific to a port that uses the OpenSSL library for cryptographic functions.

B.2.2. Header Files

B.2.2.1. TpmToOsslHash.h

```
[[TpmToOsslHash_h]]
```


B.2.2.2. TpmToOsslMath.h

[[TpmToOsslMath_h]]

B.2.2.3. TpmToOsslSym.h

[[TpmToOsslSym_h]]

B.2.3. Source Files

B.2.3.1. TpmToOsslDesSupport.c

[[TpmToOsslDesSupport]]

B.2.3.2. TpmToOsslMath.c

[[TpmToOsslMath]]

B.2.3.3. TpmToOsslSupport.c

[[TpmToOsslSupport]]

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

[[Cancel]]

C.3 Clock.c

[[Clock]]

C.4 Entropy.c

[[Entropy]]

C.5 LocalityPlat.c

[[LocalityPlat]]

C.6 NVMem.c

[[NVMem]]

C.7 PowerPlat.c

[[PowerPlat]]

C.8 Platform_fp.h

```
[[Platform_fp_h]]
```

C.9 PlatformData.h

[[PlatformData_h]]

C.10 PlatformData.c

[[PlatformData]]

C.11 PPPlat.c

[[PPPlat]]

C.12 RunCommand.c

[[RunCommand]]

C.13 Unique.c

[[Unique]]

Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

D.2 Simulator_fp.h

```
[[Simulator_fp_h]]
```


D.3 TpmTcpProtocol.h

[[TpmTcpProtocol_h]]

D.4 TcpServer.c

[[TcpServer]]

D.5 TPMCmdp.c

[[TPMCmdp]]

D.6 TPMCmds.c

[[TPMCmds]]