Trusted Platform Module Library Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.83

January 25, 2024

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

TCG

CONTENTS

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

Part 4 is provided in order to enable the explanation and demonstration, via the TCG's reference code (reproduced in Parts 3 and 4), of the functionality described in Parts 1, 2, and 3. The code in Parts 3 and 4 is not written or guaranteed to meet any level of conformance, nor does this specification require that a TPM meet any particular level of conformance. In some instances (e.g., firmware update), Part 4 cannot describe a compliant implementation. Therefore, an implementor of TPM 2.0 may decide to replace Part 4 code with vendor-specific code that enables compliance.

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.

4.1 Configuration

The TPM configuration is defined by the files TpmProfile*.h. These files can be edited in order to change the algorithms and commands supported by a TPM implementation.

4.2 Unmarshaling Code Prototype

4.2.1 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 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`

4.2.3 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.

4.2.4 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`.

4.3 Marshaling Code Function Prototypes

4.3.1 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.

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

4.3.3 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**.

4.4 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. The structure and processing of this code is complex and is provided in the code.

4.5 Part 3 Parsing

The Command / Response tables in Part 3 are reflected in command-specific data structures used by functions in this TPM 2.0 Part 4. They are:

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

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

4.6 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 Marshaling

5.1 Introduction

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).

5.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	**Comments**
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
TPM_RC
TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
                         BOOL flag)
{
    TPM_RC    result;
    result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    if(*target == TPM_RH_NULL)
    {
        if(flag)
            return TPM_RC_SUCCESS;
        else
            return TPM_RC_VALUE;
    }
    if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
      &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
            return TPM_RC_VALUE;
    return TPM_RC_SUCCESS;
}
```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
UINT16
TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

5.3 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.

5.4 Unmarshal and Marshal a Structure

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

Table xxx — Definition of TPMT_PUBLIC Structure

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

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

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

```
TPM_RC
TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
{
    TPM_RC     result;
    result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
                                      buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
                                     buffer, size, flag);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
                                   buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
                                    buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
                                         buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
                                      buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

The marshaling code for the TPMT_PUBLIC structure is:

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

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

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

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

    return result;
}
```

5.5 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;
}
```

5.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 5.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

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

Table xxx — Definition of TPM2B_EVENT Structure

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

```
TPM_RC
TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result != TPM_RC_SUCCESS)
```

```
        return result;
    // if size equal to 0, the rest of the structure is a zero buffer
    //  so stop processing
    if(target->t.size == 0)
        return TPM_RC_SUCCESS;
    if((target->t.size) > 1024)     // This check is triggered by the {:1024}
                                    // notation on ‘buffer’
        return TPM_RC_SIZE;
    result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
                                   (INT32)(target->t.size));
    if(result != TPM_RC_SUCCESS)
        return result;
    return TPM_RC_SUCCESS;
}
```

using these structure definitions:

```
typedef union {
    struct {
        UINT16        size;
        BYTE          buffer[1024];
    }             t;
    TPM2B         b;
} TPM2B_EVENT;
```

6 TPM Reference Implementation Include Files

[[tpm/include/**]]

7 TPM Reference Implementation Source Files

`[[tpm/src/**]]`

Annex A
(informative)
Implementation Dependent

A.1 Introduction

These files 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 Registry,
2) Part 2 of this specification, and
3) A purpose-built document that contains vendor-specific information in tables.
4) 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.

`[[TpmConfiguration/**]]`

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 Common Cryptographic Functionality

[[tpm/cryptolibs/common/]]**

B.3 TpmBigNum

[[tpm/cryptolibs/TpmBigNum/]]**

B.4 OpenSSL-Specific Files

B.4.1. Introduction

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

[[tpm/cryptolibs/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.

[[Platform/]]**

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.

`[[Simulator/**]]`