TSS System Level API and TPM Command Transmission Interface Specification

Family "2.0"
Level 00, Revision 01.00
26 January 2015

Contact: admin@trustedcomputinggroup.org

TCG

TCG Published

Disclaimers, Notices, and License Terms

CONTENTS

FIGURES

1 Introduction

The scope of this document is to describe all the software interfaces to the TSS System API (SAPI) and TPM Command Transmission Interface (TCTI). Both of these interfaces are part of the TPM Software Stack (TSS).

This document presents a standard for the System API and TCTI portions of the TSS; this document will be made public before the rest of the TSS specification. This document may eventually be combined into the overall TSS specification.

1.1 TSS Overview

The TSS is a software stack designed to isolate TPM application programmers from the low level details of interfacing to the TPM. The TSS consists of multiple layers: Feature API, Enhanced System API, System API, TCTI, TAB, resource manager, and TPM driver.

The TPM driver is the OS-specific driver that handles all the handshaking with the TPM and reading and writing of data to the TPM.

The resource manager manages the TPM context in a manner similar to a virtual memory manager. It swaps objects, sessions, and sequences in and out of the limited TPM onboard memory as needed. This layer is transparent to the upper layers of the TSS and is not required. However, if not implemented, the upper layers will be responsible for TPM context management.

The TPM access broker (TAB) handles multi-process synchronization to the TPM. A process accessing the TPM can be guaranteed that it will be able to complete a TPM command without interference from other competing processes.

The TPM command transmission interface (TCTI) handles all the communication to and from the lower layers of the stack. For instance, different interfaces are required for local HW TPMs, firmware TPMs, virtual TPMs, remote TPMs, and the TPM simulator. Also, there are two different interfaces to TPMs: the legacy TIS interface and the command/response buffer (CRB).

The System API (SAPI) is the lowest level interface that understands how to build command byte streams (marshalling) and decompose response byte streams (unmarshalling). The SAPI provides a bare metal (meaning very low level) software interface to the Part 3 commands in the TPM 2.0 specification.

The Enhanced System API (ESAPI) is an interface that is intended to sit directly above the System API. The primary purpose of the ESAPI is to reduce the complexity required of applications that desire to send individual “system level” TPM calls to the TPM, but that also require cryptographic operations on the data being passed to and from the TPM. In particular, applications that wish to utilize secure sessions to perform Hash-based Message Authentication Code (HMAC) operations, parameter encryption, parameter decryption, TPM command audit, and TPM policy operations could benefit from using the ESAPI.

The feature/environment API provides a higher level software abstraction to application developers. For instance, an application may want to create a key without any knowledge of the low level details. This level of abstraction will be provided by the feature and application APIs.

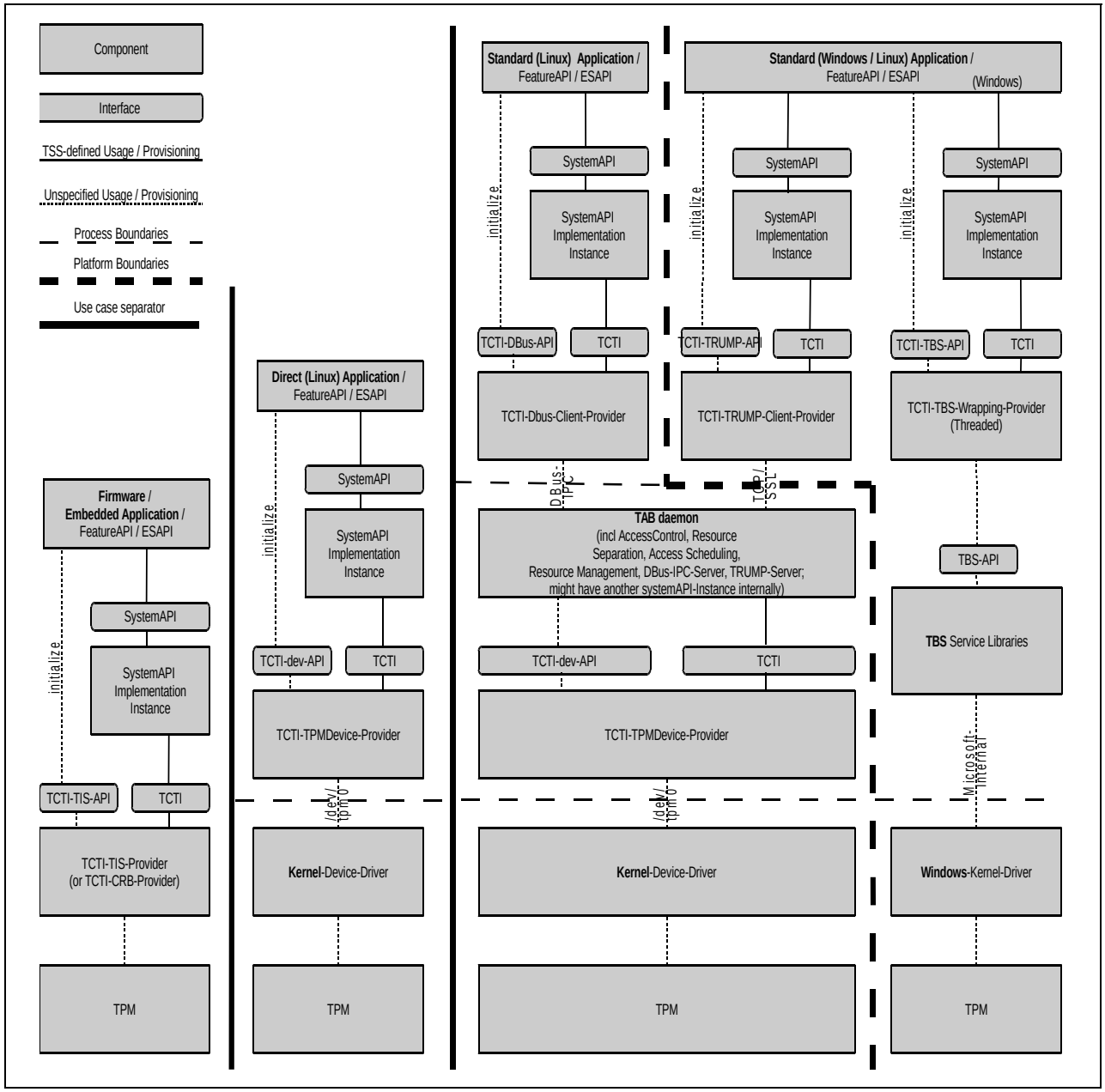

Figure 1 — Example Implementation Architectures

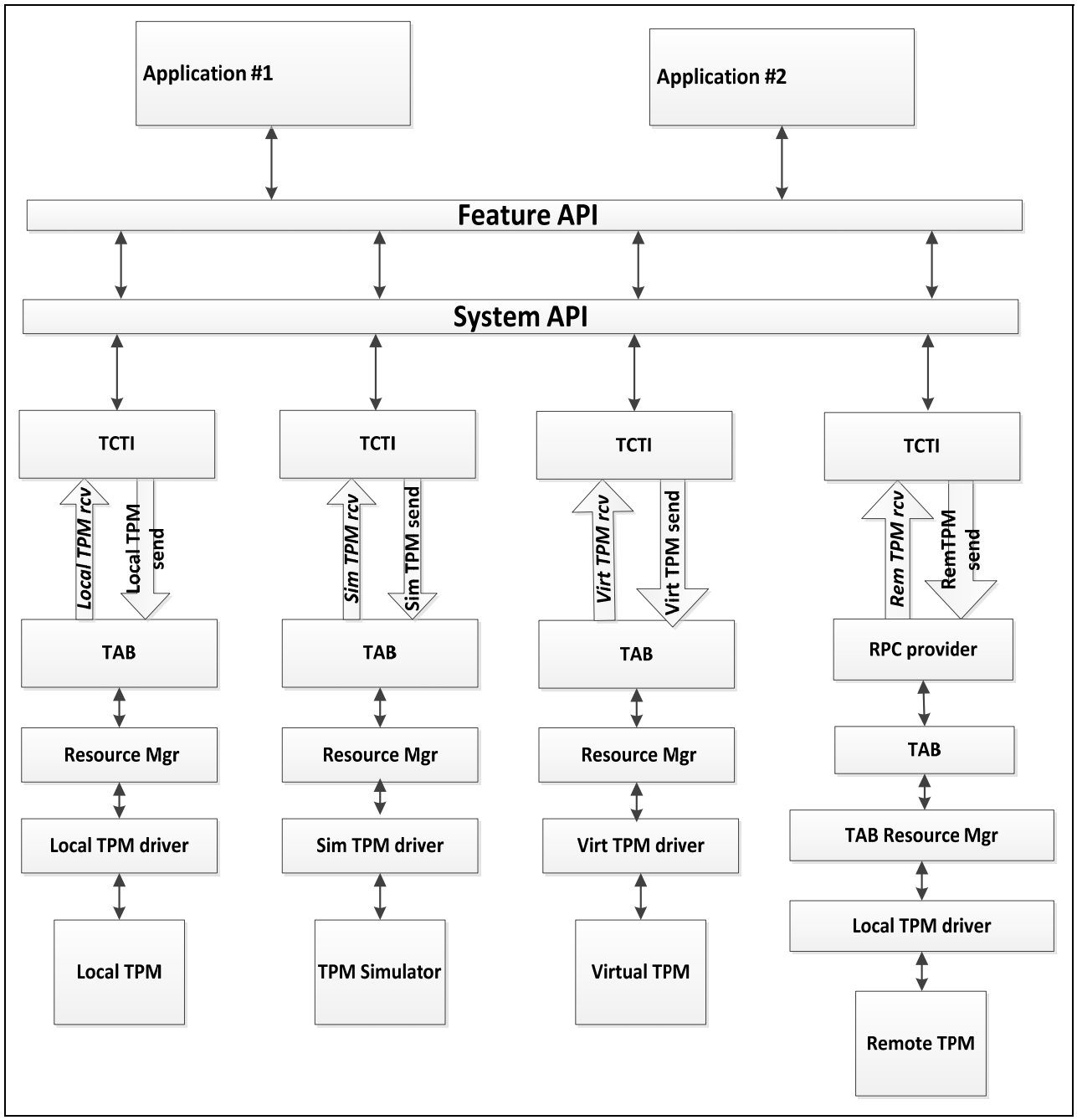

Figure 2 — TSS Stack

1.2 SAPI and TCTI Target Systems

Both of these interfaces are designed to be used in a large range of computing devices from highly embedded systems to client computers to high end servers.

2 Definition of Terms

Application programming interface or API: the software interface used to communicate between layers in the stack.

Application or application layer: Another name for the caller. See caller definition below.

Caller: in the context of this document, this is the software entity making function calls into the layer being currently discussed.

ESAPI: TSS Extended System API. This layer is intended to sit on top of the System API.

Marshalled data: this is the representation of data used to communicate with the TPM. In order to optimize data communication to and from the TPM, the smallest amount of data possible is sent to the TPM. For instance, if a structure starts with a size field and that field is set to 0, none of the other fields in the structure are sent to the TPM. Another example: if an input structure consists of a union of data structures, the marshalled representation will be the size of just the data structure selected from the union (actually the marshalled version of that structure itself). Also, the marshalled data must be in big-endian format since this is what the TPM expects.

Implementation: the source code and binary code that embodies this specification or parts of this specification.

Marshal: convert data from C structures to marshalled data.

SAPI: TSS System API

TPM2B: any TPM 2.0- data structure whose name starts with the prefix, “TPM2B_”. These structures have two fields, a UINT16 size field and another structure. The size field is the number of UINT8’s in the marshalled version of the second field. There are two main subtypes of TPM2B structures:

Complex TPM2B: a TPM2B whose second field is not an array of UINT8

Simple TPM2B: a TPM2B whose second field is an array of UINT8

TCTI or TPM command transmission interface: this is the interface used to send command to and receive responses from the TPM.

Unmarshal: convert data from marshalled format to C structures.

3 Architecture

The SAPI and TCTI are low level interfaces intended for expert applications.

Use of the SAPI requires expert knowledge of the underlying TPM 2.0 commands and architecture. The purpose of the SAPI is to enable applications to perform Part 3 commands using all possible variations of inputs to those commands and receiving all possible variations of outputs. The System API marshals inputs in C structure format to command byte streams and unmarshals responses from response byte stream format to C structure format. It uses an instance of the TCTI to do the communication with the TPM.

The TCTI abstracts callers from the device drivers and provides a common API. TCTI provides a generic interface to a wide variety of transport methods that could be used to communicate to the TPM.

4 Acronyms

ABI Application binary interface

API Application programming interface

NV Non-volatile

PCR Platform Configuration Register

TABI TPM Access Broker Interface

TCTI TPM Command Transmission Interface

TDD TPM Device Driver

TPM Trusted Platform Module

TSP TSS Service Provider

TSS TPM Software Stack

5 Overall Design Requirements

The following design requirements apply to both the SAPI and the TCTI:

These APIs shall not unnecessarily limit any system architectures that may want to use the implementations of the APIs. Some examples of the flexibility that the APIs and their implementations shall allow:

- Multiple TPMs shall be supported both on the same platform and on remote platforms.
- A single application can talk to multiple TPMs.
- The APIs shall not inhibit the use of multi-thread synchronization of accesses to the TPM nor require it.
- Should support most commercially available operating systems as well as embedded, non-OS systems.
- Should support ANSI C compilers.

The SAPI and TCTI shall not require implementations to maintain state.

The APIs shall provide “opaque-ness” with respect to their context structures by providing functions for:

- Getting and setting all necessary data.
- Returning the required size of the context structures so that the caller can properly allocate space for these structures.
- These functions isolate the application from implementation-specific or implementation version-specific changes in the size and structure of the context structures.

The APIs shall provide a way to return error codes in a manner that differentiates them from other layers’ error codes.

All interface data structures shall be allocated by the caller and not by the implementations of these APIs.

5.1 Threading Model

The TCTI and SAPI follow the same design model of a non-blocking and thread-aware library. This allows the implementation and usage of the presented APIs even in highly embedded devices and/or mainloop-driven applications that may not provide threading at all or with specific thread isolation mechanisms.

This means that unless otherwise stated, every function is supposed to be a non-blocking operation. This specification does not give exact deadlines for function executions, but they are expected to require the usual amount of cycles for non-blocking operations.

For multi-threaded application, this means that any TCTI or SAPI context may only be called from one thread at a given time. The application may implement a means to synchronize concurrent access to the same TCTI or SAPI context if desired. It is however recommended to instantiate one context per thread that needs to communicate with the TPM.

If a multi-threaded application is calling in to the TCTI or SAPI, our library does not make sure that concurrent access to the same context is secured via mutexes or other similar mechanisms inside the SAPI-module. An application that wants to use only one context globally (from all threads) will need to implement some kind of mutex itself and make sure that only one of its threads accesses the same context at the same time. This approach is however discouraged.

Instead, the recommended method is to initialize multiple SAPI-contexts, one for each thread that needs access to the TPM. The concurrent access of different threads to distinct contexts is safe and allowed in a system that includes a TAB.

6 Common Data Structures

The following data types are common between TCTI and SAPI and all other layers of the TSS. They are included in tss2_common.h.

6.1.1 Application Binary Interface (ABI) Negotiation

It is expected that multiple iterations of this specification will need to be issued in the future. Furthermore, the wide applicability of this specification and the library-nature of the TPM specification make it very likely, that multiple variations of the TSS APIs will exist in the future. In order to be able to differentiate them at runtime and provide safety of applications, an ABI negotiation shall take place inside each “initialize” call. Note that for the TCTI, ABI safety is provided in a different way.

The following ABI negotiation scheme is used at several layers of the TSS.

The TPM 2.0 library specification has implementation defined constants that can be different for different platforms. Also, the specification can change as bugs are found and fixed.

For instance, the SAPI implementation may have a different understanding of the implementation defined parameters and specification version from that of the calling application. For this reason an ABI negotiation scheme needs to be implemented.

This section describes that ABI negotiation scheme.

A TSS2_ABI_VERSION structure holds the relevant information about the ABI of a specific TSS implementation. A tssCreator value of 0x1 reflects the "standard tss" as defined by the TSS-WG for "normal systems".

This tssCreator value, 1, is targeted at interoperability. As an example, they will include "safety margins" on buffer sizes.

For vendor-specific implementations that use smaller buffer sizes, the family has a value of 0x00002 and the level holds a vendor identifier from the TCG-vendor-registry. The version field can be used by the vendor to distinguish different version.

```
typedef struct {
       UINT32 tssCreator; /* If == 1, this equals TSSWG-Interop */
                  /* IF > 0x20000000  , this equals TCG TPM capabilities Vendor-ID */
       UINT32 tssFamily;  /* Free-to-use for creator > TCG_VENDOR_FIRST */
       UINT32 tssLevel;   /* Free-to-use for creator > TCG_VENDOR_FIRST */
       UINT32 tssVersion; /* Free-to-use for creator > TCG_VENDOR_FIRST */
} TSS2_ABI_VERSION;
```

For each layer that uses ABI negotiation, a function will validate that the ABI version requested by the layer is actually supported by the layer below it. For the SAPI, the function that does this is Tss2_Sys_Initialize. If the requested ABI-Version is not supported it will return TSS2_SYS_RC_ABI_MISMATCH and set the fields to those values that would be supported.

Note that a module may support multiple versions of ABI at the same time either due to backwards compatibility or by using other appropriate techniques. In those cases, the module may accept ABI requests for multiple ABIs, but can only return one of those. It is recommended to report back either the most modern ABI or the ABI version closest to the ABI version that was requested. In case of an unsupported ABI, the module must of course reply with an ABI_MISMATCH failure.

A typical set of TSS header files for a layer should include a definition of the ABI version used throughout the layer’s header files.

```
#define TSS2_ABI_CURRENT_VERSION {.tssCreator = 0xXX, .tssFamily = 0xXXXX, tssLevel = xx,
tssVersion = xx }
```

6.1.2 Error Codes

6.1.2.1 Common Error Codes

The error type for all TSS errors is:

```
typedef UINT32 TSS2_RC;
```

The code for returning success for all TSS functions is:

```
#define TSS2_RC_SUCCESS 0
```

The shift value used to locate the TSS layer indicator is 16. This means the byte consisting of bits 23 - 16 contains the layer indicator for response codes.

```
#define TSS2_RC_LEVEL_SHIFT 16
```

6.1.2.2 Base Error Codes

Base error codes are not returned directly, but are combined with an ERROR_LEVEL to produce the error codes for each layer. The reason for these base error codes is to allow similar errors for TCTI and SAPI to have the same base code.

```
#define TSS2_BASE_RC_GENERAL_FAILURE        1 /* Catch all for all errors
                                                 not otherwise specifed */
#define TSS2_BASE_RC_NOT_IMPLEMENTED        2 /* If called functionality isn't implemented */
#define TSS2_BASE_RC_BAD_CONTEXT            3 /* A context structure is bad */
#define TSS2_BASE_RC_ABI_MISMATCH           4 /* Passed in ABI version doesn't match
                                                 called module's ABI version */
#define TSS2_BASE_RC_BAD_REFERENCE          5 /* A pointer is NULL that isn't allowed to
                                                 be NULL. */
#define TSS2_BASE_RC_INSUFFICIENT_BUFFER    6 /* A buffer isn't large enough */
#define TSS2_BASE_RC_BAD_SEQUENCE           7 /* Function called in the wrong order */
#define TSS2_BASE_RC_NO_CONNECTION          8 /* Fails to connect to next lower layer */
#define TSS2_BASE_RC_TRY_AGAIN              9 /* Operation timed out; function must be
                                                 called again to be completed */
#define TSS2_BASE_RC_IO_ERROR              10 /* IO failure */
#define TSS2_BASE_RC_BAD_VALUE             11 /* A parameter has a bad value */
#define TSS2_BASE_RC_NOT_PERMITTED         12 /* Operation not permitted. */
#define TSS2_BASE_RC_INVALID_SESSIONS      13 /* Session structures were sent, but */
                                              /* command doesn't use them or doesn't use
                                                 the specifed number of them */
#define TSS2_BASE_RC_NO_DECRYPT_PARAM      14 /* If function called that uses decrypt
                                                 parameter, but command doesn't support
                                                 decrypt parameter. */
#define TSS2_BASE_RC_NO_ENCRYPT_PARAM      15 /* If function called that uses encrypt
                                                 parameter, but command doesn't support
                                                 decrypt parameter. */
#define TSS2_BASE_RC_BAD_SIZE              16 /* If size of a paremeter is incorrect */
#define TSS2_BASE_RC_MALFORMED_RESPONSE    17 /* Response is malformed */
#define TSS2_BASE_RC_INSUFFICIENT_CONTEXT  18 /* Context not large enough */
```

```
#define TSS2_BASE_RC_INSUFFICIENT_RESPONSE 19 /* Response is not long enough */
#define TSS2_BASE_RC_INCOMPATIBLE_TCTI     20 /* Unknown or unusable TCTI version */
#define TSS2_BASE_RC_NOT_SUPPORTED         21 /* Functionality not supported. */
#define TSS2_BASE_RC_BAD_TCTI_STRUCTURE    21 /* TCTI context is bad. */
```

6.1.2.3 TCTI Error Codes

A 32-bit value is used for error codes.

All TCTI error codes will have the following in bits 23 – 16 of the error code to indicate the TSS layer that the error is coming from:

```
#define TSS2_TCTI_ERROR_LEVEL             ( 10 << TSS2_RC_LEVEL_SHIFT)
```

Bits 11 – 0 of the error code will contain a TCTI specific error as detailed below:

```
#define TSS2_TCTI_RC_GENERAL_FAILURE           ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                   TSS2_BASE_RC_GENERAL_FAILURE))
#define TSS2_TCTI_RC_NOT_IMPLEMENTED           ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                   TSS2_BASE_RC_NOT_IMPLEMENTED))
#define TSS2_TCTI_RC_BAD_CONTEXT               ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_CONTEXT))
#define TSS2_TCTI_RC_ABI_MISMATCH              ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_ABI_MISMATCH))
#define TSS2_TCTI_RC_BAD_REFERENCE             ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_REFERENCE))
#define TSS2_TCTI_RC_INSUFFICIENT_BUFFER       ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_INSUFFICIENT_BUFFER))
#define TSS2_TCTI_RC_BAD_SEQUENCE              ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL |  \
                                                    TSS2_BASE_RC_BAD_SEQUENCE))
#define TSS2_TCTI_RC_NO_CONNECTION             ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_NO_CONNECTION))
#define TSS2_TCTI_RC_TRY_AGAIN                 ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_TRY_AGAIN))
#define TSS2_TCTI_RC_IO_ERROR                 ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_IO_ERROR))
#define TSS2_TCTI_RC_BAD_VALUE                 ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_VALUE))
#define TSS2_TCTI_RC_NOT_PERMITTED             ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_NOT_PERMITTED))
#define TSS2_TCTI_RC_MALFORMED_RESPONSE        ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_MALFORMED_RESPONSE))
#define TSS2_TCTI_RC_NOT_SUPPORTED            ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_NOT_SUPPORTED))
```

6.1.2.4 SAPI Error Codes

All SAPI internal error codes will have the following in bits 23 – 16 of the error code to indicate the TSS layer that the error is coming from:

```
#define TSS2_SYS_RC_LEVEL                  8 << TSS2_RC_LEVEL_SHIFT
```

Bits 11 – 0 of the error code will contain a SAPI specific error as detailed below:

```
#define TSS2_SYS_RC_GENERAL_FAILURE           ((TSS2_RC)(TSS2_TCTI_ERROR_LEVEL | \
                                                   TSS2_BASE_RC_GENERAL_FAILURE))
```

```
#define TSS2_SYS_RC_ABI_MISMATCH                ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_ABI_MISMATCH))
#define TSS2_SYS_RC_BAD_REFERENCE               ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_REFERENCE))
#define TSS2_SYS_RC_INSUFFICIENT_BUFFER         ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_INSUFFICIENT_BUFFER))
#define TSS2_SYS_RC_BAD_SEQUENCE                ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_SEQUENCE))
#define TSS2_SYS_RC_BAD_VALUE                   ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_VALUE))
#define TSS2_SYS_RC_INVALID_SESSIONS            ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_INVALID_SESSIONS))
#define TSS2_SYS_RC_NO_DECRYPT_PARAM            ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_NO_DECRYPT_PARAM))
#define TSS2_SYS_RC_NO_ENCRYPT_PARAM            ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_NO_ENCRYPT_PARAM))
#define TSS2_SYS_RC_BAD_SIZE                    ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_SIZE))
#define TSS2_SYS_RC_MALFORMED_RESPONSE          ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_MALFORMED_RESPONSE))
#define TSS2_SYS_RC_INSUFFICIENT_CONTEXT        ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_INSUFFICIENT_CONTEXT))
#define TSS2_SYS_RC_INSUFFICIENT_RESPONSE       ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_INSUFFICIENT_RESPONSE))
#define TSS2_SYS_RC_INCOMPATIBLE_TCTI           ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_INCOMPATIBLE_TCTI))
#define TSS2_SYS_RC_BAD_TCTI_STRUCTURE          ((TSS2_RC)(TSS2_SYS_ERROR_LEVEL | \
                                                    TSS2_BASE_RC_BAD_TCTI_STRUCTURE))
```

The following level identifies errors that are just like the Part 2 error codes but come from the implementation rather than the TPM itself. This would occur if the implementation duplicated the error checking functionality that is performed by the TPM. Error codes of this type will have the following level in bits 23 – 16 of the error code to indicate the TSS layer that the error is coming from. Any function that corresponds to a Part 3 command (Tss2_Sys_XXXX_Prepare, Tss2_Sys_XXXX_Complete, and one-call) MAY return one of these error codes:

```
#define TSS2_SYS_PART2_RC_LEVEL             9 << TSS2_RC_LEVEL_SHIFT
```

6.1.2.5 TPM errors

Errors that come from the TPM will be returned unaltered to higher levels of the TSS stack. In this case, bits 15 – 12, the error layer indicator, of the error code will be set to 0.

```
#define TSS2_TPM_RC_LEVEL                   0
```

7 TPM Command Transmission Interface

The TPM Command Transmission Interface (TCTI) is used to send marshalled commands to and receive marshalled responses from the TPM (or the underlying software stack that ultimately interacts with a TPM). It is designed to handle a wide variety of transmission methods.

7.1 Intro

7.1.1 Purpose & Goal

The TPM Command Transmission Interface is designed to make different modules connectable interchangeably.

Additionally, it is possible to load these modules at runtime. An application or the Feature-API may be configurable with regards to the “TCTI-drivers” they offer to a user. The TCTI is designed specifically for these use cases and provides conventions and helpers to be runtime-loadable friendly while still allowing compile-time linking without namespace clashes.

Initialization of the TCTI interface is accomplished in a driver-specific manner and is out of scope for this specification.

7.2 TCTI data structures

All TCTI data structures are included in the following file: tss2_tcti.h.

TSS2_TCTI_POLL_HANDLE: The TCTI supports an asynchronous mode of operation for processing TPM commands. After transmission of a command, the application can request to be notified when the response is available for reception. The TSS2_TCTI_POLL_HANDLE type is used as a data type for the handles that are used when querying for this notice. Since these handles are highly platform specific, they will change depending on the used platform. For some platforms the handles are defined in the following; further handle types will be defined in future revisions of this specification. Note, that not all TCTI-drivers have support for this function.

POSIX.1-2001 / Linux:

```
typedef struct pollfd TSS2_TCTI_POLL_HANDLE;
```

Windows:

```
typedef HANDLE TSS2_TCTI_POLL_HANDLE;
```

All others (typically where async mode isn’t used), so code compiles until further specified in future revisions:

```
typedef void TSS2_TCTI_POLL_HANDLE;
```

The following are used to configure timeout characteristics:

```
#define  TSS2_TCTI_TIMEOUT_BLOCK    -1
#define  TSS2_TCTI_TIMEOUT_NONE     0
```

7.3 TCTI Context data structures

7.3.1 TCTI Context

The TCTI Context is an opaque pointer when passed in to the SAPI implementation. It is created by the caller, typically using a TCTI implementation. However, the SAPI implementation must extract several fields (typically function pointers) in order to communicate with the next lower layer.

The SAPI implementation must not write any fields of the TCTI context.

The design pattern divides the context into three areas:

- A version area
- A non-opaque area
- An opaque area

The structure definitions are defined as below.

```
typedef struct TSS2_TCTI_OPAQUE_CONTEXT_BLOB TSS2_TCTI_CONTEXT;

/* superclass to get the version */
typedef struct {
    uint64_t magic;
    uint32_t version;
} TSS2_TCTI_CONTEXT_VERSION ;

/* current version #1 known to this implementation */
typedef struct {
    uint64_t magic;
    uint32_t version;
    TSS2_RC (*transmit)( TSS2_TCTI_CONTEXT *tctiContext, size_t size,
uint8_t *command);
    TSS2_RC (*receive) (TSS2_TCTI_CONTEXT *tctiContext, size_t *size,
uint8_t *response, int32_t timeout);
    void (*finalize) (TSS2_TCTI_CONTEXT *tctiContext);
    TSS2_RC (*cancel) (TSS2_TCTI_CONTEXT *tctiContext);
    TSS2_RC (*getPollHandles) (TSS2_TCTI_CONTEXT *tctiContext,
TSS2_TCTI_POLL_HANDLE *handles, size_t *num_handles);
    TSS2_RC (*setLocality) (TSS2_TCTI_CONTEXT *tctiContext, uint8_t locality);
} TSS2_TCTI_CONTEXT_COMMON_V1;

typedef TSS2_TCTI_CONTEXT_COMMON_V1 TSS2_TCTI_CONTEXT_COMMON_CURRENT;
```

7.3.2 Function Invocation

In order to call any of these functions, an application first needs to check that a given TCTI Context has the correct version, which is a version greater or equal to the version at the time that the function was added to the common function table. Then it needs to check if the function is also implemented by the respective driver, that is, if the function pointer is non-NULL. Only then can the function be safely called. This process can be encapsulated inside a helper such as the following example:

```
#define tss2_tcti_transmit(tctiContext, size, command) \
       ((tctiContext == NULL) ? TSS2_TCTI_RC_BAD_CONTEXT: \
       (((TSS2_TCTI_CONTEXT_VERSION *)tctiContext)->version < 1) ? \
```

```
        TSS2_TCTI_RC_ABI_MISMATCH: \
(((TSS2_TCTI_CONTEXT_V1 *)tctiContext)->transmit == NULL) ? \
        TSS2_TCTI_RC_NOT_IMPLEMENTED: \
((TSS2_TCTI_CONTEXT_V1 *)tctiContext)->transmit(tctiContext, size, command))
```

A similar pattern can be implemented for all functions of TCTI.

7.3.3 Non-opaque Area

The contents of a non-opaque area are defined by the *version*. The TCG defines versions and their contents. Each newer version must contain all fields of the previous version in the same order. Function pointers must have identical parameters and return values. This permits the SAPI implementation to safely cast a TCTI context to any version not greater than the value in the *version* field.

Once the context is cast, structure members can be dereferenced and used. A typical structure member is a function pointer provided by the TCTI implementation.

7.3.3.1 Version Area

The TCTI user extracts the version area by casting the opaque context to a TSS2_TCTI_CONTEXT structure. Each context structure must begin with the same members as the TSS2_TCTI_CONTEXT structure.

The *magic* value is some unique number, perhaps simply a random number. The SAPI implementation will likely not read it. The value is a sanity check, ensuring that the lower TCTI layer receives a context that it can interpret, typically one it created.

The *version* value denotes the version of the context structure. This value must monotonically increase from older to newer versions.

A given user of the TCTI (e.g. a SAPI implementation) may support multiple version for an underlying TCTI module.

In the simplest case, it supports one version. This would not necessarily be the latest version, but rather the lowest version which contains all fields that the implementation requires. Using the lowest possible version permits the implementation to be used by back level TCTI implementations that might not implement the latest version.

In a more complex case, it supports multiple versions. This might be useful if a version is deprecated, where a function pointer has been superseded by an improved function.

7.3.3.2 Function Pointers

The SAPI implementation extracts these function pointers from the non-opaque area of the TCTI context.

7.3.3.2.1 transmit

```
TSS2_RC (*transmit)(
       TSS2_TCTI_CONTEXT           *tctiContext,
       size_t                      size,
       const uint8_t               *command
);
```

Transmits the command packet of size bytes to next layer below the caller. This function is expected to be a non-blocking operation (with respect to the given context).

Errors returned:

TSS2_TCTI_RC_NO_CONNECTION: if connection to the TPM fails

TSS2_TCTI_RC_BAD_CONTEXT: bad version and/or magic fields in the TCTI-context

TSS2_TCTI_RC_BAD_SEQUENCE: if transmit called more than once without a call to receive in between

TSS2_TCTI_RC_BAD_REFERENCE: if tctiContext or command is NULL

TSS2_TCTI_RC_BAD_VALUE: if any parameter's value is bad

NOTE: Some possible examples of this are: size out of range, size not at least 10 bytes, or size doesn't match commandSize.

7.3.3.2.2 receive

```
TSS2_RC (*receive)(
      TSS2_TCTI_CONTEXT           *tctiContext,
      size_t                      *size,
      uint8_t                     *response,
      int32_t                     timeout
);
```

Receives a response packet from the layer below the caller. . This function is expected to be a non-blocking operation (with respect to the given context).

On input, *size* is the maximum allocated byte size of *response*. On a successful return, *size* is the actual used bytes of *response*. If size is insufficient for the TPM response, the response parameter is ignored (could be NULL in this case), and the required size is returned in size with TSS2_TCTI_RC_INSUFFICIENT_BUFFER error. On input, a size bigger than 0 and a NULL pointer for the response parameter yields an error.

If *timeout* is TSS2_TCTI_TIMEOUT_BLOCK, the command is synchronous, and blocks until a response is received or a system interrupt occurs.

If *timeout* is TSS2_TCTI_TIMEOUT_NONE, the command returns immediately. *size* and *response* are updated only on a successful return.

If *timeout* is positive, the command returns after a maximum of approximately *timeout* msec.

receive always returns either the entire TPM response or a return code indicating that the response is not yet completely available. A TCTI-caller must provide the same buffer to subsequent receive-calls whenever TSS2_TCTI_RC_TRY_AGAIN is returned. A TCTI implementation that might receive partial responses can use the response buffer to assemble the partial responses in those cases.

Errors returned:

TSS2_TCTI_RC_BAD_CONTEXT: bad version and/or magic fields in the TCTI-context

TSS2_TCTI_RC_TRY_AGAIN: if a timeout or a system interrupt occurred before the complete response was received.

TSS2_TCTI_RC_INSUFFICIENT_BUFFER: if the response buffer is too small for the TPM response. In this case, the returned size indicates the size of the buffer that is needed for the response.

TSS2_TCTI_RC_IO_ERROR: Underlying IO failed.

TSS2_TCTI_RC_MALFORMED_RESPONSE: if an implementation chooses to validate the internals of a response and finds some error in the response. One example would be if less than 6 bytes are received which means the response size cannot be determined.

TSS2_TCTI_RC_BAD_REFERENCE: if tctiContext, size is NULL, or response is NULL and *size != 0

TSS2_TCTI_RC_BAD_VALUE: If timeout is negative but not -1 (TSS2_TCTI_TIMEOUT_BLOCK)

TSS2_TCTI_RC_BAD_SEQUENCE: if receive called again after first successful receive or if transmit not called first

7.3.3.2.3 finalize

```
TSS2_RC (*finalize) (
       TSS2_TCTI_CONTEXT            *tctiContext
);
```

This function performs any actions required when a TCTI connection is terminated and invalidates the TCTI Context. The TCTI Context cannot be used for subsequent operations after this call. This function should be called whenever a TCTI Context is not needed anymore. Afterwards the TCTI Context memory can be freed. If tcitiContext is NULL, no operation is performed.

Errors returned:

TSS2_TCTI_RC_BAD_CONTEXT: bad version and/or magic fields in the TCTI-context

7.3.3.2.4 cancel

```
TSS2_RC (*cancel) (
       TSS2_TCTI_CONTEXT            *tctiContext
);
```

This function causes the TCTI layer to cancel the command; in some TCTI implementations this may include sending the TPM cancel command. This command can only be called between transmit and receive calls.

NOTE: After calling cancel, receive still needs to be called to get the return code, because the TPM may choose to complete the command in response to the cancel command. The only way for the caller to know whether the command completed or was cancelled is to check the return code.

Errors returned:

TSS2_TCTI_RC_NO_CONNECTION: if connection to the TPM fails.

TSS2_TCTI_RC_BAD_SEQUENCE: if not called between transmit and receive.

TSS2_TCTI_RC_BAD_REFERENCE: if tctiContext is NULL

TSS2_TCTI_RC_BAD_CONTEXT: bad version and/or magic fields in the TCTI-context

7.3.3.2.5 getPollHandles

```
TSS2_RC (*getPollHandles) (
       TSS2_TCTI_CONTEXT            *tctiContext,
       TSS2_TCTI_POLL_HANDLE        *handles,
       size_t                       *num_handles
);
```

This function retrieves the handles that can be used for polling or select. This function returns a set of handles that can be used to poll for incoming responses from the underlying software stack or TPM. The type for these handles is platform specific and defined separately in the declaration of TSS2_TCTI_POLL_HANDLE. In order to query the number of handles that a TCTI module needs to have monitored, the application may pass NULL for handles; in this case, it returns the number of handles. In pseudo-code this could be:

```
do {    *getPollHandles (tctiContext, NULL,&num);
    TSS2_TCTI_POLL_HANDLES handles[num];
    *getPollHandles (tctiContext, &handles[0], &num);
    poll(&handles[0], num, -1);     /* Platform specific syscall */
    ret = *receive(tctiContext, &buffer_size, &buffer[0]);
while (ret == TSS2_TCTI_RC_TRY_AGAIN);
```

Errors returned:

TSS2_TCTI_RC_BAD_REFERENCE: if tctiContext, handles, or num_handles is NULL

TSS2_TCTI_RC_BAD_CONTEXT: bad version and/or magic fields in the TCTI-context

7.3.3.2.6 setLocality

```
TSS2_RC (*setLocality) (
       TSS2_TCTI_CONTEXT           *tctiContext,
       uint8_t                     locality
);
```

Set the locality for the TPM.

The locality cannot be changed between transmit and receive

Errors returned:

TSS2_TCTI_RC_BAD_REFERENCE: if tctiContext is NULL

TSS2_TCTI_RC_NO_CONNECTION: if connection to the next lower layer fails

TSS2_TCTI_RC_BAD_SEQUENCE: if locality is changed between transmit and receive calls

TSS2_TCTI_RC_BAD_VALUE if TCTI can’t support the locality

TSS2_TCTI_RC_NOT_PERMITTED if the change in locality is not permitted.

TSS2_TCTI_RC_NOT_SUPPORTED if the lower layer doesn’t support changing localities at all.

TSS2_TCTI_RC_BAD_CONTEXT: bad version and/or magic fields in the TCTI-context

7.4 Compatibility

This design anticipates that the non-opaque area might change over time, with new functions added and existing functions deprecated or even deleted (set to NULL). The section lists a few such use cases.

The SAPI implementation might receive an old but still completely useful context. Assuming it has a structure definition for this version (which is why it was recommended that the implementation use the oldest suitable structure), it casts to that version and continues.

7.4.1 Old and Not Useful Version

The SAPI implementation can receive an old TCTI context version that it cannot use because it requires a function pointer not available in the old version. The implementation detects the down level version (by detecting that it's less than required version) and returns an error.

The TCTI implementation must be updated to support the new requirements.

7.4.2 New but Useful Version

If the SAPI implementation receives a newer version than the one compiled in, it can cast the TCTI context to its down level version and use existing structure members. The cast is safe because new members are added at the end of the structure and existing members are never removed.

The implementation cannot access new members, but it was not coded to use them anyway.

7.4.3 New and Not Useful Version

The implementation might receive a newer context version where structure members that it requires have been superseded by newer members. For example, the TCTI implementation might provide a new function pointer with a different parameter list.

The TCTI implementation flags this situation by setting an older, now unimplemented function pointer to NULL. After the cast, the SAPI implementation detects the NULL. It cannot continue because a function it requires is no longer available.

The SAPI implementation must be updated to use the new function pointer.

7.4.4 New Version with Deprecated Functions

If the SAPI implementation receives a context version with both deprecated and recommended function pointers, it can handle it one of two ways.

If it was compiled to only handle a back level context, it will cast and use only the deprecated function pointers.

If it was compiled to handle several context versions, it can cast to a more recent version and use the recommended function pointers.

7.5 Opaque Area

The SAPI implementation cannot access the opaque area of the TCTI context. The TCTI implementation can add any implementation specific fields as needed to the opaque area.

The TCTI is expected to cast the context to its implementation specific context type based on the version number. Since the TCTI both creates and consumes the context, no incompatibility is expected.

The TCTI implementation may also validate that the magic value is as expected. This sanity check ensures that the pointer passed in is indeed a TCTI context known to the TCTI implementation.

8 SAPI

The System API provides a programming interface that provides access to all the functionality of the TPM while performing the necessary marshaling and unmarshaling operations on the parameters of TPM commands. This enables software running on top of the System API to operate using C structure and host-native data representations only.

8.1 Overall functionality

SAPI is designed to be used wherever low level calls to the TPM functions are made: firmware, BIOS, applications, OS, etc.

The System API, as a low level interface, is targeted towards expert applications.

Its purpose is to provide access to the full range of the TPM 2.0 capabilities and to do this in a way that makes the caller's job as easy as possible. It is designed to be used in a wide range of computing devices, from very low end, i.e. highly embedded, systems to high end servers.

8.2 Design requirements

The requirements for the SAPI are:

1. The API SHALL provide access to all features of all TPM 2.0 commands as defined in part 3 of the TPM 2.0 specification.
2. The SAPI implementation SHALL NOT require cryptographic functionality or session management functionality. Its main purpose is to provide access to TPM 2.0 commands; it only supports those plus a minimal set of required management functions. All other functionality is out of scope for the System API.
3. The SAPI SHALL NOT unnecessarily limit any system architectures that may want to use the System API and its implementation. Some specifics:
    a. The System API SHALL NOT preclude systems which don't need HMACs or cpHash pre-calculations nor SHALL it burden such systems with unnecessary overhead related to HMACs or cpHash calculations.
    b. The API SHALL be configurable with respect to the handling of return data required by applications. For instance, many commands return values that the caller may not want or need and some of these are code intensive to return in unmarshalled form. NULL buffer pointers passed in for particular parameters can be used by the API to signal the implementation that these return values aren't needed.
4. An implementation of the SAPI SHALL support both asynchronous and synchronous calls for all TPM 2.0 commands.
5. For each TPM 2.0 command, an implementation of the SAPI SHALL support a command-specific Tss2_Sys_XXXX_Prepare function call that provides calling applications with the information needed to calculate a cpHash for the command. This cpHash is used by the application for calculating HMAC authorizations and pre-calculating policy hashes which will be used in creating objects that will use policy authorizations.
6. The API SHALL NOT require the implementation to allocate memory for any input or output data structures. It is the calling application's responsibility to allocate any memory needed.
7. All parameters of SAPI functions SHALL be in native-endian format. This means that the SAPI implementation will do any endian conversion required for both inputs and outputs.

8. The SAPI implementation SHALL perform formatting, marshalling, and unmarshalling tasks so that the caller needs as little knowledge of the TPM 2.0 command format as possible. Marshalling of input and unmarshalling of output data is performed by the SAPI.
9. The SAPI implementation SHALL return all “unhandled” error codes from lower layers in the TSS stack to the SAPI caller without alteration. This means that TPM error codes that aren’t “handled” are returned to the caller unaltered. Typically, the only way TPM error codes would be handled is by an underlying resource manager. An example would be when an attempt is made to load an object but there isn’t enough space in the TPM. In this case, the resource manager will flush some object(s) to provide enough space and then re-try the load operation. The error code from the first load operation that failed will not be returned since it was handled by the resource manager.

8.3 Design rules

In order to best achieve the System API requirements, the following design rules for the System API were developed:

1. SAPI functions that execute TPM 2.0 commands typically execute one TPM 2.0 command.

   Note: one exception to this is if underlying layers such as the resource manager do TPM 2.0 commands in order to fulfil their role.

2. As much as possible, the System API will mimic the TPM 2.0, Part 3 command schematics and Part 1 command and response layout diagrams. Function input and output parameters are ordered in the way they appear in the Part 3 command schematics and variable names match as much as possible.

   NOTE: This will help programmers understand the code and easily correlate it to the specification.

3. Since memory for input and output parameters are provided by the caller, some design rules result:
   a. All output parameters will require a pointer to be passed to the API.
   b. In order to minimize the stack memory requirements, inputs that are not simple data types or bit fields will be passed in as pointers. This is different from the TPM 1.2 TSS; the reason for this change is to minimize stack usage for systems with very little memory. TPM 2.0 uses much larger structures than TPM 1.2 making this even more important.
   c. Buffers for the input command byte stream, output response byte stream, cpBuffer, and rpBuffer are not required to be allocated by the implementation. NOTE: For heapless implementations, they are expected to be located within the opaque SAPI context structure.
4. The System API implementation will do as much work for the caller as possible. Some examples of this:
   a. The commandSize field for all commands is calculated dynamically by the implementation.
   b. Output parameters will be unmarshalled into C structures before being returned to the caller so that the caller can read fields out of them in a straightforward manner.
5. The only function pointers used by the SAPI code are to TCTI functions. No function pointers are used to call from the System API implementation to “helper” functions to calculate cpHash or HMAC or to perform any other crypto or session management tasks. Instead a layer above the SAPI explicitly makes those calls. This provides the most flexibility possible to the ESAPI or

whatever other layer is directly above the SAPI. For instance, it may want to use different HMAC helper functions depending on what actions are being performed.

8.4 Architecture

1. Each SAPI function that corresponds to a Part 3 command, takes the following inputs:
    a. A pointer to a SAPI context structure that is used to maintain any state required. This structure is allocated by the caller and initialized via Tss2_Sys_Initialize.
    b. A group of handles and command and/or response parameters to the TPM:
        i. Inputs:
            1. Input parameters are in “TSS System API” form to make the caller’s job easier. This means that C structures will be used as inputs.
            2. All inputs to the TPM that are data structures are input as pointers to those data structures the System API.
        ii. Outputs:
            1. A group of pointers to buffers, one for each possible output data item or structure. If any output pointer is NULL, the output is not required by the caller and the implementation will not do any work to provide that output to the caller.
    c. All commands that aren’t restricted to the TPM_ST_NO_SESSIONS command will be capable of handling between 0 and 3 sessions or authorizations on input and output. Sessions and authorizations are input through a data structure that:
        i. Identifies the number of sessions or authorizations that are in use
        ii. Contains all the data for each session/authorization.
2. The SAPI implementation marshals the input parameters before sending them to the TPM. This includes the following:
    a. Endian conversion if necessary.
    b. Tag, command size (computed by the implementation after marshalling is complete), command code, and handles are filled in.
    c. Fills in authorization size followed by marshalled authorizations in the order they were received the input array.
3. To send the data to the next lower layer, the SAPI implementation calls a function pointer in the structure pointed to by the tctiContext member of the SAPI context structure. This function pointer, transmit, is passed a pointer to the input command buffer and the size of the input buffer.
4. To receive the response, the SAPI implementation calls a function pointer in the structure pointed to by tctiContext member of the SAPI context structure. This function pointer, receive, is passed a pointer to the output response buffer, the size of the output buffer, and a timeout parameter.
5. The System API implementation checks for errors in the transmission to or reception of the data from the TPM. Handling of these errors is the responsibility of the caller.
6. If an error has occurred, the System API implementation returns this error code to the application. No more processing of the returned data occurs in this case.
7. If the TPM command completed successfully, for all outputs that received a non-null output pointer, the System API implementation:

a. Unmarshals the output into a C structure, which includes converting the endianness when necessary.

b. Copies the data to the structure pointed to by the output parameter pointer.

8. The System API implementation unmarshals the response authorization areas into C structures.

9. TSS System API assumes that all required initialization is done before any System API functions are called. Specifically:

a. Underlying TAB instantiations, resource manager(s), and device driver interfaces are initialized.

b. Applications that call into the System API know which TPMs are available and tell the System API how to communicate to the particular TPMs that are used by means of the TCTI context structures.

c. Figure 3 below is a drawing of this. Purple outlined blocks and lines are done at initialization time. The definition of initialization time is implementation specific and out of scope for this specification.

NOTE: The sequence could be done as follows:

a) Some system process initializes the TSS stack, TAB instantiation(s), including resource manager(s), and TPM 2.0 driver(s). How and when this is done is implementation-specific with the only requirement being that this must be done before System API functions are called.

b) Application(s) start.

c) TCTI context structure(s) are initialized with pointers to send/receive functions. How and when this is done is implementation-specific with the only requirement being that this must be done before System API functions are called.

d) Applications call System API functions, initializing the cmdContext structures; part of this initialization includes setting a pointer to point to the TCTI structure which contains the correct send/receive function pointers and interface name.

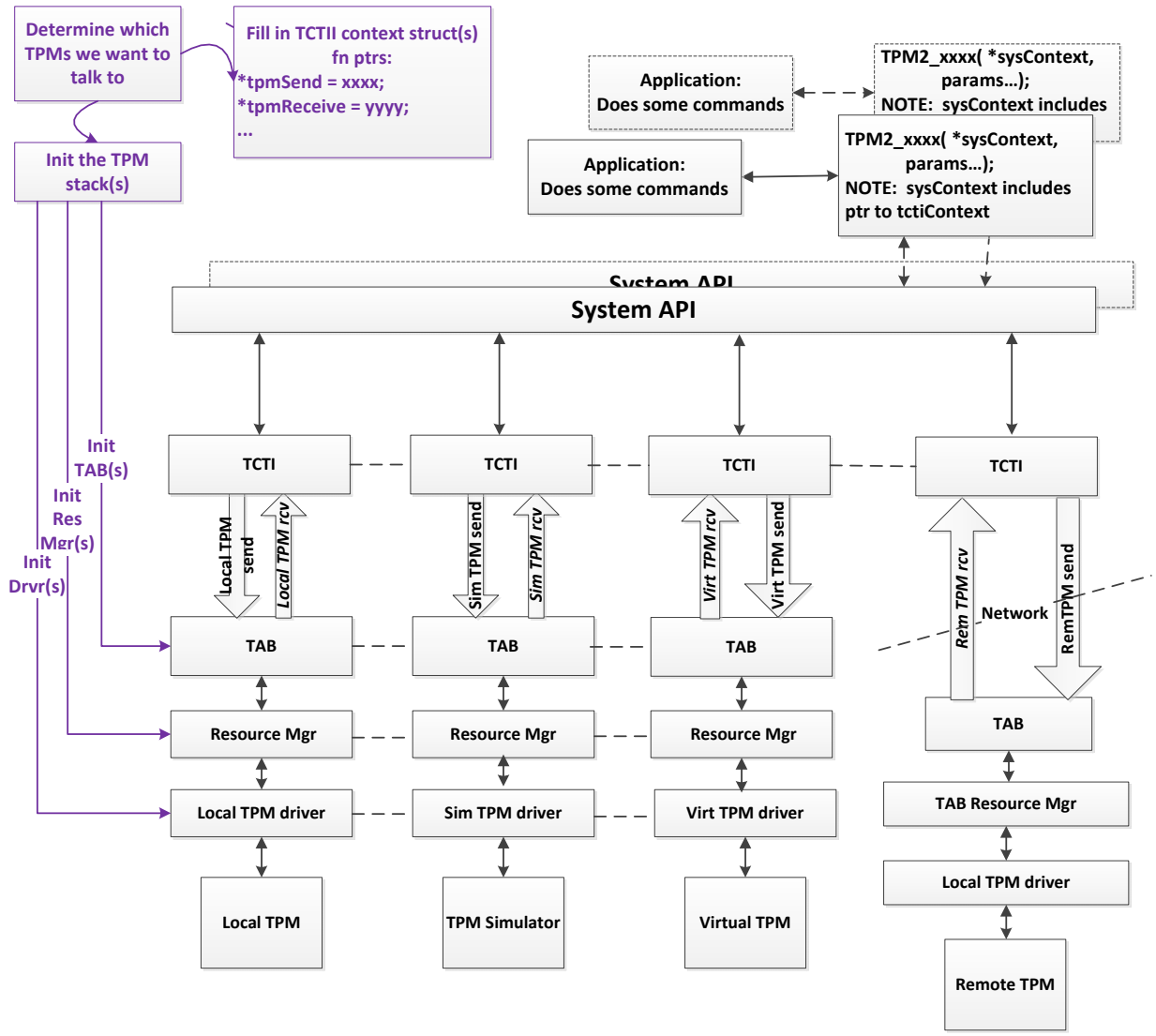

Figure 3 — Low level stack details

8.5 SAPI data structures

All SAPI data structures and functions are defined in the following header file: tss2_sys.h.

8.5.1 Part 2 Data Types

See TPM 2.0 specification, Part 2, for a description of data types used in Part 3 specific SAPI functions. These data types are defined in the following header file: tss2_tpm2_types.h.

The TPM specification version used for this document is the 01.15 version.

For TPM algorithms, the version of the algorithm registry used for this document is the 01.16 version.

In order to guarantee forward compatibility, this specification assumes the following maximums:

```
#define MAX_RSA_KEY_BITS        4096
```

```
#define MAX_AES_KEY_BITS        256
#define MAX_ECC_KEY_BITS        256
#define IMPLEMENTATION_PCRS     32
#define MAX_ACTIVE_SESSIONS     0x00ffffff
#define MAX_LOADED_SESSIONS     0x00ffffff
#define MAX_LOADED_OBJECTS      0x00ffffff
#define MAX_CONTEXT_SIZE        3072
#define MAX_NV_BUFFER_SIZE      2048
#define PRIVATE_VENDOR_SPECIFIC_BYTES 0
```

MAX_CAP_BUFFER MAX is set to the maximum of 1024 and the sum of:

- sizeof(TPM_CAP)
- sizeof(UINT32)
- the maximum of the following:
    - sizeof(TPMS_ALG_PROPERTY)
    - sizeof(TPM_HANDLE)
    - sizeof(TPM_CC)
    - sizeof(TPMS_TAGGED_PROPERTY)
    - sizeof(TPMS_TAGGED_PCR_SELECT)
    - sizeof(TPM_ECC_CURVE)

In order to know the smallest possible number of bytes in a TPMS_PCR_SELECT structure, a SAPI implementation must query the TPM for the number of platform PCRs by performing a GetCapability command for the TPM_PT_PCR_SELECT_MIN property.

8.5.2 sysContext structure

An opaque context structure that is used (individually) by SAPI implementation to store state information as well as e.g. buffers in case of heapless implementations.

```
typedef struct _TSS2_SYS_OPAQUE_CONTEXT_BLOB TSS2_SYS_CONTEXT;
```

External to SAPI implementations, there should never be a definition of struct _TSS2_SYS_OPAQUE_CONTEXT_BLOB. Each implementation will provide their own versions of this blob and an application can determine the necessary size during initialization.

This structure is an opaque structure that should never be defined outside of a SAPI implementation. It contains all buffers and state required during the execution of a Part 3 command.

8.5.3 Command and response session structures

This structure is used to set the session data that is passed to and returned from the SAPI Part 3 functions.

Input structure for command authorization area(s).

```
typedef struct {
       uint8_t              cmdAuthsCount;
       TPMS_AUTH_COMMAND    **cmdAuths;
} TSS2_SYS_CMD_AUTHS;
```

cmdAuthsCount is the used size of the *cmdAuths* array, representing the number of authorization areas in the command stream. *cmdAuths* is an array of pointers to single TPMS_AUTH_COMMAND structures.

```
typedef struct {
       uint8_t              rspAuthsCount;
       TPMS_AUTH_RESPONSE   **rspAuths;
} TSS2_SYS_RSP_AUTHS;
```

rspAuthsCount is the size of the *rspAuths* array, representing the number of authorization areas in the response stream. *rspAuths* is an array of pointers to single TPMS_AUTH_RESPONSE structures.

For a single command, *cmdAuthsCount* must be the same as *rspAuthsCount*, since the TPM always returns the same number of authorizations as it receives unless there is an error.

8.6 Command Parameters

The parameters for Part 3 command-specific functions, Tss2_Sys_XXXX_Prepare, Tss2_Sys_XXXX_Complete, and one-call, are the C structures as specified in section 6.5.1 above.

In order to guarantee binary compatibility between applications and SAPI implementations built with different compiler tool chains, wherever the TPM specification Part 2 specifies a bit field, the API expects a 32 bit unsigned integer where bit 0 is the least significant bit in host endianness representation.

8.6.1 System API Parameter Rules

All parameters and handles in authorization areas and Part 3 specific functions will be passed in as:

1. Simple parameters if they are command inputs or fields in the authorization regions AND they are one of the following types:
    - Base types as described in Part 2 of the TPM specification. This includes types like TPM_HANDLE for instance.
    - Bit fields
2. Pointers to the parameter types, if they are anything other than the above types OR if they are response parameters. For input-parameters these pointer can point to const memory structures.

8.7 SAPI Function APIs (by Category)

Within the SAPI functions, there are two types of functions:

- TPM 2.0 Part 3 command specific functions
- Generic functions which are not specific to particular Part 3 commands

The SAPI function APIs are split into the following groups, each of which may be Part 3 specific or not:

- Command Context Allocation Functions: all non-specific.
- Command Context Setup Functions: all non-specific.
- Command Preparation Functions: Tss2_Sys_XXXX_Prepare calls are Part 3 specific, all others are not specific.
- Command Execution Functions: “one-call” functions are Part 3 specific, all others are not specific.
- Command Completion Functions: Tss2_Sys_XXXX_Complete functions are Part 3 specific, all others are not specific.

- Part 3 command-specific functions will be described as templates in these sections: Tss2_Sys_XXXX_Prepare, one-call (Tss2_Sys_XXXX), and Tss2_Sys_XXXX_Complete.

The function prototypes for all of these functions are in the tss2_sys.h header file.

8.7.1 Command Context Allocation Functions

8.7.1.1 Tss2_Sys_GetContextSize

```
size_t  Tss2_Sys_GetContextSize(
      size_t                    maxCommandResponseSize
);
```

Returns the required size for the opaque SAPI command context. The caller must allocate a context of at least this size.

If *maxCommandResponseSize* is 0, Tss2_Sys_GetContextSize returns a size guaranteed to handle any TPM command and response supported by this specification. The caller may specify a non-zero *maxCommandResponseSize* corresponding to the maximum expected command and response size in order to save memory within the TSS2_SYS_CONTEXT SAPI context.

NOTE: For a heapless implementation of the SAPI, the returned value will be larger than maxCommandResponseSize. How much larger depends on the System API implementation. For instance, if an implementation uses two independent buffers for transmit and receive, then the size returned could be 2X the passed in size plus whatever extra memory is needed by the System API for context.

NOTE: If the caller constrains the size, a subsequent command may then return TSS2_SYS_RC_INSUFFICIENT_CONTEXT if the actual size of a command or response exceeds the size specified in this function. For an error forming the command, the caller may be able to start over with a larger context. For a response, error handling is more complicated because the response may be in transit. It is therefore recommended to use maxCommandResponseSize set to 0 for context allocation.

8.7.1.2 Tss2_Sys_Initialize

```
TSS2_RC Tss2_Sys_Initialize(
      TSS2_SYS_CONTEXT          *sysContext,
      size_t                    contextSize,
      TSS2_TCTI_CONTEXT         *tctiContext,
      TSS2_ABI_VERSION          *abiVersion
);
```

This function must be called once on a newly allocated sysContext. It need not be called to reuse a sysContext. The systemAPI may perform getCapability calls or anything else required for enabling workarounds of TPM bugs. These should be done during the Tss2_Sys_Initialize call and these should not be done at other times.

The contextSize parameter specifies the size of the memory area reserved for the sysContext. It is expected to be the same size returned from a corresponding Tss2_Sys_GetContextSize call (see above).

The tctiContext parameter holds the pointer to an initialized TCTI Context that will be used by subsequent calls to communicate with the TPM. It can retrieved later using Tss2_Sys_GetTctiContext but cannot be altered during the lifetime of one systemAPI context. If tctiContext is NULL, no calls other than Tss2_Sys_XXXX_Prepare can be called using the sysContext. The Tss2_Sys_Initialize function SHOULD check that the essential TCTI function pointers (transmit and receive) are not NULL and return an error code otherwise.

Note: a given TCTI Context should only be used with one SAPI context simultaneously in order to prevent collisions between transmit and receive calls that are using different SAPI contexts. For this reason, it is recommended to have a 1-to-1 relation between SAPI and TCTI contexts.

The *abiVersion* parameter holds information about the version and revision of this specification as used by the application. The SAPI implementation will check if it is compatible with this version. If not, it will return an error and overwrite this parameter with the ABI version it supports. The corresponding variable can be initialized via the TSS2_ABI_VERSION_CURRENT definition (see the TCTI Context section). This pointer could be NULL if the System API library is being statically linked and the developer is completely sure that the versions match. In this case, no check of abiVersion is performed.

Errors Returned:

TSS2_SYS_RC_ABI_MISMATCH: if input ABI doesn't match that of the System API implementation.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_VALUE: if any parameter has a bad value.

TSS2_SYS_RC_BAD_REFERENCE: if any of *sysContext* is a NULL pointer.

TSS2_SYS_RC_BAD_TCTI_STRUCTURE: if the implementation checks the TCTI function pointers and any of the essential ones (transmit and receive) are set to NULL.

Any TPM or TCTI errors that could result from GetCapability calls that are made to get TPM version info.

TSS2_SYS_RC_INCOMPATIBLE_TCTI: unknown or unusable TCTI version.

8.7.1.3 Tss2_Sys_Finalize

```
TSS2_RC Tss2_Sys_Finalize(
      TSS2_SYS_CONTEXT            *sysContext
);
```

This function should be called before freeing a sysContext. If sysContext is NULL, no operation is performed.

Errors Returned:

Reserved for future use.

8.7.1.4 Tss2_Sys_GetTctiContext

```
TSS2_RC Tss2_Sys_GetTctiContext(
      TSS2_SYS_CONTEXT            *sysContext,
      TSS2_TCTI_CONTEXT           **tctiContext
);
```

This function returns the pointer to the tctiContext that was passed in during the Tss2_Sys_Initialize call. This function can, for example, be used to retrieve the tctiContext associated with a given sysContext before freeing the sysContext in order to either reuse or free the associated tctiContext.

Errors Returned:

TSS2_SYS_RC_BAD_REFERENCE: if *sysContext* or *tctiContext* are NULL pointers.

8.7.2 Command Preparation Functions

8.7.2.1 Tss2_Sys_XXXX_Prepare

```
TSS2_RC Tss2_Sys_XXXX_Prepare(
      TSS2_SYS_CONTEXT            *sysContext,
                                  inHandles,
                                  inParams
);
```

This is a template for a Tss2_Sys_XXXX_Prepare function. There is one of these per Part 3 TPM command. The number and types of input parameters and handles are defined in Part 3 of the TPM specification.

All passed in information is saved in the *sysContext* for use during subsequent commands.

If any command parameter (after the handles) is a TPM2B:

- If the TPM2B parameter is NULL, the implementation marshals a TPM2B with a size of 0.

  NOTE: The TPM often uses this pattern of setting the TPM2B size field to 0 for optional parameters.

- If the TPM2B parameter is not NULL:
  - If the TPB2B is a simple TPM2B, the TPM2B size field indicates the used size of the UINT8 array. The implementation marshals the size and buffer into the byte stream. The implementation will not read beyond the used size.

    NOTE: The size may be zero.

    NOTE: For the first command parameter, the UINT8 array may be encrypted before or after Tss2_Sys_XXXX_Prepare. To encrypt after Tss2_Sys_XXXX_Prepare, see Tss2_Sys_GetDecryptParam.

  - If the TPM2B is a complex TPM2B, the TPM2B size field will be ignored.

    NOTE: The implementation will calculate the size and marshal the TPM2B second parameter based on its data type.

    NOTE: Ignoring the size field permits the application to use a response parameter unmodified as an input to a subsequent command. Since the size is ignored, if the caller wants a complex TPM2B to be marshaled with a size of 0, it should be passed in as NULL.

NOTE: For the first command parameter that is a TPM2B, the following also apply:

If the first command parameter is non-NULL, the implementation will marshal it. After Tss2_Sys_XXXX_Prepare, the parameter may be encrypted. See Tss2_Sys_GetDecryptParam.

If the first command parameter is NULL, then a zero sized TPM2B is inserted for that command parameter. The caller may (and normally should) add the parameter after the Tss2_Sys_XXXX_Prepare. See Tss2_Sys_SetDecryptParam.

The Tss2_Sys_XXXX_Prepare command resets the *sysContext* and makes it ready for next flow of command functions. After any Tss2_Sys_XXXX_Prepare, previously set authorizations are made unavailable and Tss2_Sys_SetCmdAuths must be called again.

NOTE: As an example of why it is required that Tss2_Sys_XXXX_Prepare reset the sysContext, suppose the following sequence occurs: Tss2_Sys_XXXX_Prepare, Tss2_Sys_SetCmdAuths, Tss2_Sys_XXXX_Prepare . If the second Tss2_Sys_XXXX_Prepare is done with different

parameters than the first Tss2_Sys_XXXX_Prepare, the authorizations previously set by Tss2_Sys_SetCmdAuths are no longer valid after the second Tss2_Sys_XXXX_Prepare. To handle this as cleanly as possible and avoid burdening the implementation with the overhead of checking the parameters for changes, after any Tss2_Sys_XXXX_Prepare (whether the parameters have changed or not), previously set authorizations are made unavailable and Tss2_Sys_SetCmdAuths must be called again.

Errors Returned:

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_VALUE: if bad values are used for parameters.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext or a parameter that is a pointer is set to NULL.

TSS2_SYS_RC_BAD_SEQUENCE: if called between Tss2_Sys_ExecuteAsync and Tss2_Sys_ExecuteFinish.

The function MAY report marshalling errors in the same format as the TPM would, with the error level set to TSS2_SYS_PART2_RC_LEVEL.

8.7.2.2 Tss2_Sys_GetDecryptParam

```
TSS2_RC Tss2_Sys_GetDecryptParam(
      TSS2_SYS_CONTEXT            *sysContext,
      size_t                      *decryptParamSize,
      const uint8_t               **decryptParamBuffer
);
```

This function returns the size and pointer to a buffer corresponding to the first marshaled TPM2B command parameter as *decryptParamSize* and *decryptParamBuffer*. If the first command parameter passed to Tss2_Sys_XXXX_Prepare was NULL, *decryptParamSize* is 0 and *decryptParamBuffer* is unspecified.

This function must be called after Tss2_Sys_XXXX_Prepare and before the Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync.

The application must not write to the returned *decryptParamBuffer* and this buffer may only be considered readable until the next invocation of any function that uses the same sysContext. If any other SAPI call is made to the same *sysContext* a previously retrieved *decryptParamBuffer* contains undefined data and Tss2_Sys_GetDecryptParam must be called again.

If this function is called after a Tss2_Sys_SetDecryptParam, the newly set parameter is returned.

The intent of this call is to provide the size and location of the parameter to be encrypted by the caller in a decrypt session after the parameter has been marshaled by Tss2_Sys_XXXX_Prepare. After calling this, the encrypted result can be set by calling Tss2_Sys_SetDecryptParam.

NOTE: If encryption is performed, it must be performed after Tss2_Sys_XXXX_Prepare and before Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync. It should typically be called before Tss2_Sys_GetCpBuffer is called for the cpHash calculation, since the cpHash must be calculated using encrypted version of this parameter.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if not called after Tss2_Sys_XXXX_Prepare and before Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync.

TSS2_SYS_RC_NO_DECRYPT_PARAM: if called when *sysContext* is set for a command that doesn't have a decrypt parameter.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_REFERENCE: if any of the inputs is a NULL pointer.

8.7.2.3 Tss2_Sys_SetDecryptParam

```
TSS2_RC Tss2_Sys_SetDecryptParam(
      TSS2_SYS_CONTEXT            *sysContext,
      size_t                      decryptParamSize,
      const uint8_t               *decryptParamBuffer
);
```

If the first command parameter is a TPM2B type, this function sets the size and buffer of the TPM2B. If the first parameter is not a TPM2B, this function returns an error.

If Tss2_Sys_XXXX_Prepare received a non-NULL first command parameter TPM2B, this function replaces the TPM2B buffer and the TPM2B size field must match *decryptParamSize*. If the Tss2_Sys_XXXX_Prepare received a NULL first command parameter, this function updates the size and inserts the TPM2B buffer.

This function must be called after Tss2_Sys_XXXX_Prepare and before Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync.

NOTE: It should typically be called before Tss2_Sys_GetCpBuffer is called in preparation for cpHash calculation, since the cpHash calculation has to be done using the encrypted version of this parameter.

NOTE: The intent of this call is to set the first command TPM2B parameter when the caller has marshaled and encrypted the parameter.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if not called after Tss2_Sys_XXXX_Prepare and before Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_REFERENCE: if *sysContext* or *decryptParamBuffer* is null.

TSS2_SYS_RC_BAD_SIZE: if the first TPM2B parameter was not NULL at Tss2_Sys_XXXX_Prepare and *decryptParamSize* does not equal the marshaled size.

TSS2_SYS_RC_NO_DECRYPT_PARAM: if called when *sysContext* is set for a command that doesn't have a decrypt parameter.

8.7.2.4 Tss2_Sys_GetCpBuffer

```
TSS2_RC Tss2_Sys_GetCpBuffer(
      TSS2_SYS_CONTEXT            *sysContext,
      size_t                      *cpBufferUsedSize,
      const uint8_t               **cpBuffer
);
```

This function returns the *cpBuffer*, a pointer to the marshaled command parameters, and the number of used bytes in the cpBuffer.

This function can only be called after a Tss2_Sys_XXXX_Prepare and before a Tss2_Sys_Execute, Tss2_Sys_ExecuteAsync, or one-call function.

The application must not write to the returned *cpBuffer* and this buffer may only be considered readable until the next invocation of any function that uses the same *sysContext*. If any other SAPI call is made to the same *sysContext* a previously retrieved cpBuffer contains undefined data and Tss2_Sys_GetCpBuffer must be called again.

NOTE: It is typically used for calculating the cpHash value for command authorization. It is typically called after an optional Tss2_Sys_SetDecryptParam call (in conjunction with encryption) and before a Tss2_Sys_SetCmdAuthorization call.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if not called after Tss2_Sys_XXXX_Prepare and before Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync.

TSS2_SYS_RC_BAD_REFERENCE: if any NULL pointer is passed in.

8.7.2.5 Tss2_Sys_SetCmdAuths

```
TSS2_RC Tss2_Sys_SetCmdAuths(
      TSS2_SYS_CONTEXT            *sysContext,
      const TSS2_SYS_CMD_AUTHS    *cmdAuthsArray
);
```

This function saves the authorization data to the sysContext.

cmdAuthsArray-> cmdAuthsCount indicates the number of authorizations to add, the number of pointers in the cmdAuthsArray->cmdAuths array. Pointers must point to valid TPMS_AUTH_COMMAND structures.

If this command causes authorizations to be added, the command tag will be set to TPM_ST_SESSIONS.

NOTE: This command is optional for Part 3 commands that don't require any authorization sessions. If it is not called, the command tag defaults to TPM_ST_NO_SESSIONS.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if not called after Tss2_Sys_XXXX_Prepare and before Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext or cmdAuthsArray are NULL

TSS2_SYS_RC_BAD_VALUE: if one of the pointers in cmdAuthsArray->cmdAuths[cmdAuthsArray->cmdAuthsCount] is NULL and cmdAuthsArray->cmdAuthsCount != 0.

TSS2_SYS_RC_INVALID_SESSIONS: MAY be returned if the command cannot take authorizations.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.3 Command Execution Functions

8.7.3.1 Tss2_Sys_ExecuteAsync

```
TSS2_RC Tss2_Sys_ExecuteAsync(
      TSS2_SYS_CONTEXT            *sysContext
```

```
);
```

Calls the TCTI transmit function pointer to send the TPM command stream. It does not call the receive function.

This function is expected to return quickly.

This function can only be called once after a Tss2_Sys_XXXX_Prepare. It is typically called when all the necessary command data has been set (this may mean after Tss2_Sys_SetDecryptParam and Tss2_Sys_SetCmdAuths have been called). After this call, only the Tss2_Sys_ExecuteFinish and Tss2_Sys_GetTctiContext functions can be called on the same sysContext.

Errors Returned:

TSS2_SYS_RC_INVALID_SESSIONS: if command cannot take the number of authorizations specified by cmdAuthsArray->cmdAuthsCount.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_SEQUENCE: if called before Tss2_Sys_XXXX_Prepare or if, after the most recent Tss2_Sys_XXXX_Prepare, one of the following functions has been called: Tss2_Sys_ExecuteAsync, Tss2_Sys_Execute, one-call, or Tss2_Sys_ExecuteFinish

TSS2_SYS_RC_BAD_REFERENCE: if sysContext is a NULL pointer.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.3.2 Tss2_Sys_ExecuteFinish

```
Tss2_Sys_ExecuteFinish(
      TSS2_SYS_CONTEXT            *sysContext,
      int32_t                     timeout
);
```

Calls the receive function pointer to receive the response stream.

This function can only be called after a Tss2_Sys_ExecuteAsync and only until it does not respond with TSS2_TCTI_RC_TRY_AGAIN anymore.

If the timeout (in milliseconds) is:

- positive: return after the timeout, indicating whether the response was received
- 0: return immediately, indicating whether the response was received
- -1: return after the response is received

Errors Returned:

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_TCTI_RC_TRY_AGAIN: if timeout occurs or if SIGINT occurred and time was set to -1.

TSS2_SYS_RC_INSUFFICIENT_RESPONSE if the response does not contain at least a tag, response size, and response code.

TSS2_SYS_RC_MALFORMED_RESPONSE: if any kind of malformed response is received

TSS2_SYS_RC_BAD_REFERENCE: if sysContext is a NULL pointer.

TSS2_SYS_RC_BAD_SEQUENCE: if not called immediately after Tss2_Sys_ExecuteAsync. Exception: can be called again if TSS2_TCTI_RC_TRY_AGAIN was received

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

TSS2_SYS_RC_BAD_VALUE: if timeout is an unspecified value, i.e. a negative number other than -1

TPM return code if response is received correctly.

8.7.3.3 Tss2_Sys_Execute

```
TSS2_RC Tss2_Sys_Execute(
      TSS2_SYS_CONTEXT            *sysContext
);
```

Equivalent to Tss2_Sys_ExecuteAsync followed by Tss2_Sys_ExecuteFinish with a timeout of -1.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if called after Tss2_Sys_ExecuteAsync, Tss2_Sys_Execute, one-call, and Tss2_Sys_ExecuteFinish, and before Tss2_Sys_XXXX_Prepare

All error codes that can be returned by Tss2_Sys_ExecuteAsync and Tss2_Sys_ExecuteFinish excluding all cases of TSS2_SYS_RC_BAD_SEQUENCE in the Tss2_Sys_ExecuteAsync and Tss2_Sys_ExecuteFinish functions.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext is a NULL pointer.

8.7.3.4 Tss2_Sys_XXXX

```
TSS2_RC Tss2_Sys_XXXX(
      TSS2_SYS_CONTEXT            *sysContext,
                                  inHandles,
      TSS2_SYS_CMD_SESSIONS       *cmdAuthsArray,
                                  inParams,
                                  *outHandles,
                                  *outParams,
      TSS2_SYS_RSP_SESSIONS       *rspAuthsArray
);
```

Above is a template for a “one-call” function. The number and types of the input handles and parameters, the number and types of the pointers to output handles and parameters, and the presence or absence of the cmdAuthsArray and rspAuthsArray parameters is defined by the commands as described in Part 3 of the TPM specification.

There is one of these per TPM command. This command can be used for the following cases:

1. Sending a command that never takes authorizations.
2. Sending a command can take authorizations, but doesn’t need to in a specific instance of a call to that command.
3. Sending a command with a simple password authorization.
4. This command can do any complex authorization as well as pre-marshalled and pre-encrypted first TPM2B payload.
    a. For use with a decrypt session, the session attributes decrypt flag will be used to tell the code that the 1st input parameter, if it’s a TPM2B, is already marshalled.
    b. For use with an encrypt session, the session attributes encrypt flag will be used to tell the code that the 1st output parameter, if it’s a TPM2B, is not to be unmarshalled.

NOTE: In the above points a and b, “marshalled” refers to the contents of the TPM2B, not the size; the size after the call will still be in host-endian order.

Similar to the Tss2_Sys_XXXX_Prepare command, this command resets the sysContext and makes it ready for next flow of command functions. To execute a single Part 3 command, this function can be used in conjunction with Tss2_Sys_XXXX_Prepare, Tss2_Sys_GetCpbuffer, Tss2_Sys_GetCommandCode, and Tss2_Sys_GetRpBuffer functions; no other functions can be combined with the one-call function to send a single TPM command. Specifically this means:

Any context created by calls to Tss2_Sys_XXXX_Prepare, Tss2_Sys_GetDecryptParam, Tss2_SysSetDecryptParam, or Tss2_Sys_SetCmdAuths will be invalidated by a subsequent call to Tss2_Sys_XXXX.

If called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish and before Tss2_Sys_XXXX_Complete, the returned parameters will be lost since Tss2_Sys_XXXX resets the context.

NOTE: this is acceptable for commands that don’t return parameters or if the returned parameters aren’t needed.

Tss2_Sys_GetRspAuths, Tss2_Sys_XXXX_Complete, Tss2_Sys_GetEncryptParam, and Tss2_Sys_SetEncryptParam cannot be called after a call to Tss2_Sys_XXXX and before a call to Tss2_Sys_XXXX_Prepare.

The rpBuffer will need to be preserved after a Tss2_Sys_XXXX call so that a subsequence Tss2_GetRpBuffer call will work correctly.

If any command parameter (after the handles) is a TPM2B:

- If the TPB2B is a simple TPM2B, the TPM2B size field indicates the used size of the UINT8 array. The implementation marshals the size and buffer into the byte stream. The implementation will not read beyond the used size.

  NOTE: The size may be zero.

  NOTE: For the first command parameter, the UINT8 array may be encrypted before the one-call command.

- If the TPM2B is a complex TPM2B, the TPM2B size field will be ignored.

  NOTE: The implementation will calculate the size and marshal the TPM2B second parameter based on its data type.

NOTE: For the first command parameter, if it is a TPM2B, the following also applies: If the session attributes decrypt flag is set, the first command parameter is assumed to be pre-marshalled and encrypted.

If any response parameter is a TPM2B:

If the response parameter is a simple TPM2B:

- o On the call to the one-call function, its size parameter must be the caller allocated size of the array.
- o Before returning from the one-call function, the implementation will write the used size of the array.
- o The used size is unmarshalled from the response stream. If the used size is greater than the caller allocated size, this function returns TSS2_SYS_RC_INSUFFICIENT_BUFFER.

NOTE: If the caller reuses the TPM2B, the size must be set back to the caller allocated before the next call to Tss2_Sys_XXXX_Complete or the one-call function.

- If any parameter is a complex TPM2B:
    - In the call to the one-call function, its size parameter MUST be zero.
    - Before returning from the one-call function, its size parameter will be the unmarshalled version of the size of the TPM2B's UINT8 array as returned from the TPM.

If the first response output parameter is a complex TPM2B, and the second parameter of the TPM2B is encrypted (as indicated by the encrypt flag in the sessions attributes):

Then the response parameter cannot be unmarshalled until it is decrypted. In this case the one-call function treats the complex TPM2B like a simple TPM2B, meaning it unmarshalls the size, but not the payload. What gets returned is really not the same as the structure type specified in Part 3; a helper function will be required to decrypt the payload and then unmarshal the contents into the specified C structure.

Errors Returned:

TSS2_SYS_RC_INVALID_SESSIONS: if command cannot take or return the number of authorizations specified by cmdAuthsArray ->cmdAuthsCount.

TSS2_SYS_RC_INSUFFICIENT_BUFFER: if any of the output parameters does not provide enough buffer space.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if there is not enough internal memory in the context.

TSS2_SYS_RC_INSUFFICIENT_RESPONSE if the response does not contain at least a tag, response size, and response code.

TSS2_SYS_RC_MALFORMED_RESPONSE: if any kind of malformed response is received

TSS2_SYS_RC_BAD_SEQUENCE if called after___Tss2_Sys_ExecuteAsync and before Tss2_Sys_ExecuteFinish.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext is NULL or an input parameter that is a pointer is NULL.

TSS2_SYS_RC_BAD_VALUE: if bad values are used for parameters.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

NOTE: even if a SIGINT occurs, this function doesn't return until it completes. There is no TSS2_TCTI_RC_TRY_AGAIN returned by this function

8.7.4 Command Completion

8.7.4.1 Tss2_Sys_GetCommandCode

```
TSS2_RC Tss2_Sys_GetCommandCode(
      TSS2_SYS_CONTEXT            *sysContext,
      UINT8                       (*commandCode)[4]
);
```

This function gets the command code for the command. The command code is returned as an array of bytes in big endian order. This array can be used for calculating the cpHash for a command or rpHash for a response.

The command code returned is valid from one Tss2_Sys_XXXX_Prepare or one-call function call to the next Tss2_Sys_XXXX_Prepare or one-call function call.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if called before Tss2_Sys_XXXX_Prepare or one-call function is called.

TSS2_SYS_RC_BAD_REFERENCE: If sysContext or *commandCode* are NULL.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.4.2 Tss2_Sys_GetRspAuths

```
TSS2_RC Tss2_Sys_GetRspAuths(
      TSS2_SYS_CONTEXT            *sysContext,
      TSS2_SYS_RSP_AUTHS          *rspAuthsArray
);
```

This function gets the response authorization data from the sysContext. If an element in the rspAuthArray->rspAuths array is NULL, that authorization is skipped in the sysContext. rspAuthArray->rspAuthsCount must equal the number of response authorizations, even if one or more is skipped.

The rspAuthsArray->rspAuths buffers are not read by the SAPI implementation.

This function can only be called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish and before the next Tss2_Sys_XXXX_Prepare.

Errors Returned:

TSS2_SYS_RC_INVALID_SESSIONS: if response does not return the number of authorizations specified by rspAuthsArray->rspAuthsCount.

TSS2_SYS_RC_INVALID_SESSIONS: MAY be returned if response does not return the same number of authorizations as were sent in the command.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_SEQUENCE: if one of the following is true:

- Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish returned anything other than TPM_RC_SUCCESS
- If not called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish and before the next Tss2_Sys_XXXX_Prepare.
- If called for a command that can never take authorizations

TSS2_SYS_RC_MALFORMED_RESPONSE: if any kind of malformed response is received.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext or rspAuthsArray are NULL

TSS2_SYS_RC_BAD_VALUE: if rspAuthsArray.authsCount is 0

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.4.3 Tss2_Sys_XXXX_Complete

```
TSS2_RC Tss2_Sys_XXXX_Complete(
      TSS2_SYS_CONTEXT            *sysContext,
                                  *outHandles,
                                  *outParams
);
```

This is a template for a complete function. There is one Tss2_Sys_XXXX_Complete function per Part 3 TPM command. The number and types of pointers to output parameters and of the handles are defined in Part 3 of the TPM specification.

This function is not implemented for commands that do not return any parameters or handles.

This function unmarshals the response parameters and response handles from a previously executed command.

If the caller does not require a certain parameter to be returned, it may pass in NULL for any of the response handles or parameters and these values will not be returned.

This function must only be called after Tss2_Sys_Execute or Tss2_Sys_ExecuteAsync and before the next Tss2_Sys_XXXX_Prepare call.

If any parameter is a simple TPM2B:

On the call to Tss2_Sys_XXXX_Complete, its size parameter must be the caller allocated size of the array.

On the return from Tss2_Sys_XXXX_Complete, the implementation will write the used size of the array.

The used size is unmarshaled from the response stream. If the used size is greater that the caller allocated size, returns TSS2_SYS_RC_INSUFFICIENT_BUFFER.

NOTE: If the caller reuses the TPM2B, the size must be set back to the caller allocated size before the next call to Tss2_Sys_XXXX_Complete or one-call function.

If any parameter is a complex TPM2B:

- In the call to Tss2_Sys_XXXX_Complete, its size parameter MUST be zero.
- On the return from Tss2_Sys_XXXX_Complete, its size parameter will be the unmarshalled version of the size of the TPM2B's UINT8 array as returned from the TPM.

If the first response output parameter is a complex TPM2B, and the second parameter of the TPM2B is encrypted, then the caller has two options:

- The caller can decrypt the parameter before Tss2_Sys_XXXX_Complete is called with first the parameter non-NULL. See Tss2_Sys_GetEncryptParam and Tss2_Sys_SetEncryptParam.
- The caller can call Tss2_Sys_XXXX_Complete with the first parameter NULL. The implementation will not unmarshal it. The caller must decrypt and unmarshal the parameter. See Tss2_Sys_GetEncryptParam.

Errors Returned:

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_SEQUENCE: if one of the following is true:

- Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish returned anything other than TPM_RC_SUCCESS
- If not called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish and before the next Tss2_Sys_XXXX_Prepare.

TSS2_SYS_RC_MALFORMED_RESPONSE: if any kind of malformed response is received.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext or one of the handle pointer parameters is NULL.

The function MAY report unmarshalling errors in the same format as the TPM would, with the error level set to TSS2_SYS_PART2_RC_LEVEL.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.4.4 Tss2_Sys_GetEncryptParam

```
TSS2_RC Tss2_Sys_GetEncryptParam(
      TSS2_SYS_CONTEXT            *sysContext,
      size_t                      *encryptParamSize,
      const uint8_t               **encryptParamBuffer
);
```

If the first response parameter is a TPM2B type, this function returns the size of the buffer and pointer to the buffer of the marshaled TPM2B. If the first parameter is not a TPM2B, returns TSS2_SYS_RC_NO_ENCRYPT_PARAM.

This function MUST only be called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish function and before the next Tss2_Sys_XXXX_Prepare or one-call. Typically, it will be called between Tss2_Sys_GetRpBuffer and Tss2_Sys_XXXX_Complete, since the rpHash must contain the encrypted value before Tss2_Sys_XXXX_Complete is called. If Tss2_Sys_XXXX_Complete is called without having decrypted the parameter, Tss2_Sys_XXXX_Complete may fail with an unmarshalling error.

The application must not write to the returned encryptParamBuffer and this buffer may only be considered readable until the next invocation of any function that uses the same sysContext. If any other SAPI call is made to the same sysContext a previously retrieved encryptParamBuffer contains undefined data and Tss2_Sys_GetEncryptParam must be called again.

The intent of this call is to provide the size and readable buffer of the parameter to be decrypted by the caller in an encrypted session.

After calling this, the decrypted result can be set by calling Tss2_Sys_SetEncryptParam; this allows the Tss2_Sys_XXXX_Complete call to properly unmarshal the result.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if one of the following is true:

- If Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish returned anything other than TPM_RC_SUCCESS
- If not called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish and before the next Tss2_Sys_XXXX_Prepare.

TSS2_SYS_RC_NO_ENCRYPT_PARAM: if called when sysContext is set for a command that doesn't have an encrypt response parameter.

TSS2_SYS_RC_MALFORMED_RESPONSE: if any kind of malformed response is received

TSS2_SYS_RC_BAD_REFERENCE: if any of the inputs are NULL.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.4.5 Tss2_Sys_SetEncryptParam

```
TSS2_RC Tss2_Sys_SetEncryptParam(
      TSS2_SYS_CONTEXT            *sysContext,
      size_t                      encryptParamSize,
      const uint8_t               *encryptParamBuffer
);
```

If the first response parameter is a TPM2B type, this function sets the size and buffer of the TPM2B. The TPM2B size field must match encryptParamSize.

This function must only be called after Tss2_Sys_Execute or Tss2_Sys_ExecuteFinish and before the next Tss2_Sys_XXXX_Prepare or one-call.

The intent of this call is to set the first response TPM2B parameter's UINT8 array to the decrypted value after the caller has decrypted the parameter and before the response parameter is unmarshaled using Tss2_Sys_XXXX_Complete.

It is typically called after the rpHash calculation, since the rpHash calculation uses the encrypted version of this parameter.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if not called after Tss2_Sys_GetEncryptParam and before Tss2_Sys_XXXX_Complete.

TSS2_SYS_RC_INSUFFICIENT_CONTEXT: if for any reason there's not enough memory.

TSS2_SYS_RC_BAD_REFERENCE: if sysContext or encryptParambuffer is null

TSS2_SYS_RC_BAD_VALUE, or if is encryptParamSize size too small.

TSS2_SYS_RC_BAD_SIZE: if the first TPM2B parameter size field does not equal encryptParamSize.

TSS2_SYS_RC_NO_ENCRYPT_PARAM: if called when sysContext is set for a response that doesn't have an encrypt response parameter.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

8.7.4.6 Tss2_Sys_GetRpBuffer

```
TSS2_RC Tss2_Sys_GetRpBuffer(
       TSS2_SYS_CONTEXT            *sysContext,
       size_t                      *rpBufferUsedSize,
       const uint8_t               **rpBuffer
);
```

This function returns a pointer to the rpBuffer, a pointer to the marshaled response parameters, and the used rpBuffer bytes after command execution.

This function MUST only be called after Tss2_Sys_Execute, Tss2_Sys_ExecuteFinish, or one-call function and before the next Tss2_Sys_XXXX_Prepare.

The application must not write to the returned rpBuffer and this buffer may only be considered readable until the next invocation of any function that uses the same sysContext. If any other SAPI call is being made to the same sysContext a previously retrieved rpBuffer contains undefined data and Tss2_Sys_GetRpBuffer must be called again.

It is typically used for calculating the rpHash value for response authorization. It is typically called before an optional Tss2_Sys_GetEncryptParam /Tss2_Sys_SetEncryptParam or after a Tss2_Sys_XXXX_Complete.

Errors Returned:

TSS2_SYS_RC_BAD_SEQUENCE: if Tss2_Sys_Execute, Tss2_Sys_ExecuteFinish, or one-call function returned anything other than TPM_RC_SUCCESS.

TSS2_SYS_RC_MALFORMED_RESPONSE: if any kind of malformed response is received.

TSS2_SYS_RC_BAD_REFERENCE: if any of the inputs are NULL.

TSS2_TCTI_RC_BAD_CONTEXT: if the Tss2_Sys_Initialize call to create the sysContext was called with a NULL pointer for tctiContext.

9 Appendices

The following are all informative.

9.1 SAPI ABI Negotiation Pseudo Code

```
/* A pseudo-code example for how to use this */
applicationExample()
{
       /* first initialize the TCTi-Layer (simplified) */

       /* Call DefaultInitialize with NULL to get the size.
       Tss2_Tcti_DefaultInitialize(NULL, &tct_size);

       // Allocate space.
       tctContext = alloc(tct_size);

       /* Now, actually do the initialization.
       Tss2_Tcti_DefaultInitialize(tctContext, &tct_size);

       sysContext = alloc(sys_size = Tss2_Sys_GetContextSize(-1));
       /* then initialize the SystemAPI; get the
          values for abi-family and abi-level from
          the tss2_sysapi.h file */
       TSS2_ABI_VERSION currentAbi = TSS2_ABI_CURRENT_VERSION;
       /* Alternatively but not recommended
       TSS2_ABI_VERSION currentAbi =
       {
              .family = TSS2_ABI_FAMILY,
              .level = TSS2_ABI_LEVEL,
              ...
       };
       */
       ret = Tss2_Sys_Initialize(sysContext, sys_size, tctContext,
                     &currentAbi);

       if (ret == TSS2_ERROR_SYS_ABI_MISMATCH) {
              fprintf(stderr, "ERROR: Mismatch between application's \
ABI-Version and systemAPI's ABI-Version.\n");
              fprintf(stderr, "SystemAPI supports %d.%d.%d.%d\n",
                            currentAbi.creator, currentAbi.family,
                            currentAbi.level, currentAbi.version);
              exit (1);
       } else if (ret != TSS2_SUCCESS) {
              /* Handle other errors */
       }

       /* Use the sysContext, etc etc etc... */
```