GLOBALPLATFORM™

THE STANDARD FOR MANAGING APPLICATIONS ON SECURE CHIP TECHNOLOGY

GlobalPlatform Device Technology

TEE Client API Specification

Version 1.0

Public Release

July 2010

Document Reference: GPD_SPE_007

Table of Contents

1. Introduction

This specification defines a communications API for connecting *Client Applications* running in a rich operating environment with security related *Trusted Applications* running inside a *Trusted Execution Environment* (TEE). For the purposes of this document a TEE is expected to be a trusted environment within the main device system-on-a-chip, which complements traditional security environments such as a UICC SIM card, although this is not a requirement of the API.

1.1. Audience

This document is suitable for software developers implementing:

- Client Applications running within the rich operating environment and which use Trusted Applications
- Trusted Applications running inside the TEE which need to expose an externally visible interface to Client Applications
- the TEE and the communications infrastructure required to access it

As this API is also the base layer upon which higher level protocols can be built, it will also be of interest to developers of future specifications providing these high level APIs built on top of the TEE Client API.

1.2. References

This section includes technical and informative references used by this specification.

1.2.1. Normative References

Standard / Specification	Description	Ref
ISO/IEC 9899:1999	ISO C Standard for C99	[1]
RFC4122	A Universally Unique IDentifier (UUID) URN Namespace	[2]
RFC2119	Key words for use in RFCs to Indicate Requirement Levels	[3]

Table 1-1: Normative References

1.2.1. Informative References

Document	Description	Ref
OMTP Advanced Trusted Environment TR1	Open Mobile Terminal Platform (OMTP) Advanced Trusted Environment TR1 [1]	[4]
GPD/STIP 2.3	GPD/STIP 2.3, Mobile Profile [2]	[5]

Table 1-2: Informative References

1.3. Terminology and Definitions

The following table defines the expressions used within this specification that use an upper case first letter in each word of the expression. Expressions within this document that use a lower case first letter in each word take the common sense meaning.

[1] OMTP TR1 Document: http://www.omtp.org/Publications/Display.aspx?Id=24ad518b-6dba-4155-ad51-3143bd43a234
[2] Global Platform GPD/STIP: http://www.globalplatform.org/specificationsdevice.asp

Term	Definition
Client Application	An application running outside of the Trusted Execution Environment making use of the TEE Client API to access facilities provided by Trusted Applications inside the Trusted Execution Environment.
Command	A single remote function invocation; the Client Application accessing a function provided by the Trusted Application. Commands are issued inside an established Session.
Implementation	A specific instantiation of all of the technology which exists underneath the TEE Client API and upon which its behavior depends. Typically the behavior of an Implementation will depend on the rich operating system, the TEE, the Trusted application, and the hardware platform which is in use.
Memory Reference	An Operation Parameter that is either a Registered Memory Reference or a Temporary Memory Reference.
Operation Parameter	One of the parameters passed in an Operation Payload. It may be either a Memory Reference or a Value Parameter.
Registered Memory Reference	A region of a Shared Memory buffer being shared within a single Operation Parameter
Session	A Session represents a logical connection between a Client Application and a specific Trusted Application. Sessions are established within initialized TEE Contexts, and act as containers for Commands.
Shared Memory	A block of Client Application memory space which is shared with a Trusted Application running inside the security environment. In some implementations, this may be directly mapped memory, enabling zero-copy data transfer.
TEE Context	A TEE Context represents a logical connection between a Client Application and an entire TEE. TEE Contexts are containers for Sessions.
Temporary Memory Reference	A buffer of memory temporarily shared for the duration of an Operation.
Trusted Application	An application running inside a Trusted Execution Environment which exports security related functionality to Client Applications outside of the trusted environment.
Trusted Execution Environment	A Trusted Execution Environment (TEE) is an environment which runs alongside a rich operating system and provides security services to that rich environment. There are multiple technologies which can be used to implement a TEE, and the level of security achieved varies accordingly.
Value Parameter	An Operation Parameter that carries a small amount of raw data (two 32-bit integers).

Table 1-3: Terminology and Definitions

1.4. Abbreviations and Notations

Abbreviation	Meaning
ABI	Application Binary Interface
API	Application Programming Interface
OMTP	Open Mobile Terminal Platform
RFU	Reserved for Future Use
SIM	Subscriber Identity Module
TEE	Trusted Execution Environment
UICC	Universal Integrated Circuit Card

Table 1-4: Abbreviations and Notations

1.5. Conventions

Throughout this document, normative requirements are highlighted by use of capitalized key words as described below.

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as described in RFC2119 [3]:

MUST - This word means that the definition is an absolute requirement of the specification.

MUST NOT - This phrase means that the definition is an absolute prohibition of the specification.

SHOULD - This word means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

SHOULD NOT - This phrase means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

MAY - This word mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

2. Overview

This specification defines a communications API for connecting *Client Applications* running in a rich operating environment with security related *Trusted Applications* running inside a *Trusted Execution Environment* (TEE). For the purposes of this document a TEE is expected to be a trusted environment within the main device system-on-a-chip, which complements traditional security environments such as a UICC SIM card, although this is not a requirement of the API. A TEE provides an execution environment with security capabilities, which are either available to Trusted Applications running inside the TEE or exposed externally to Client Applications. A TEE may, for example, host a GPD/STIP runtime [5], but may also be based on other technologies such as a small operating system executing native code applications.

See the Open Mobile Trusted Platform (OMTP) Advanced Trusted Environment TR1 specification [4] for a requirements analysis of Trusted Execution Environments in mobile devices.

2.1. Standardization Scope

Instead of trying to standardize a single monolithic API which covers a significant proportion of the interactions between a Client Application and the TEE-hosted functionality, the approach of the Global Platform standardization effort is modular. The TEE Client API covered by this specification concentrates on the interface to enable efficient communications between a Client Application and a Trusted Application running inside the TEE. Higher level standards and protocol layers may be built on top of the foundation provided by the TEE Client API – for example, to cover common tasks such as secure storage, cryptography, and run-time installation of new Trusted Applications – but these interfaces are outside of the scope of this specification.

2.2. The TEE Client API Architecture

The relationship between the major system components described in this specification are outlined in the block architecture below (Figure 2-1).

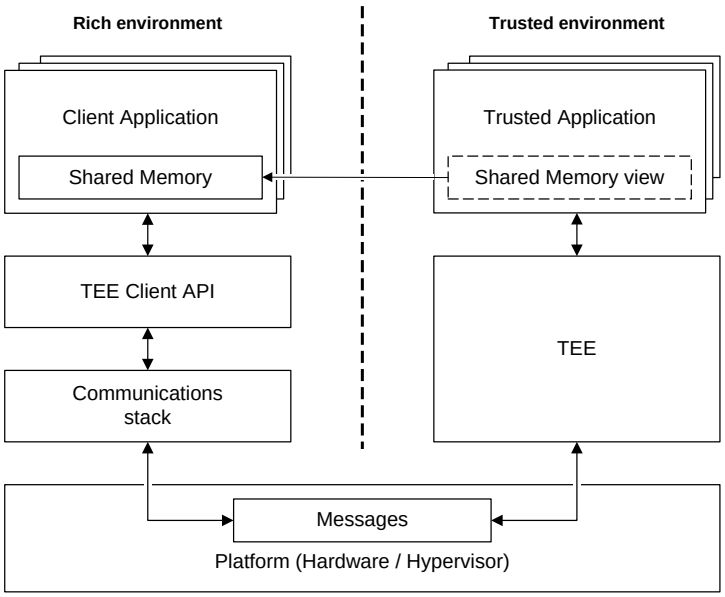

Figure 2-1: TEE Client API System Architecture

Some *implementation-defined* support is required to provide separation between the rich environment and the TEE. The mechanisms used to achieve this, and the level of security these mechanisms provide, are outside of the scope of this specification.

Within the trusted environment this specification identifies two distinct classes of component: the hosting code of the TEE itself, and the Trusted Applications which run on top of it. There is no definition of the expected implementation of these blocks in this specification; they are only used as logical concepts inside this document.

Within the rich environment this specification identifies three distinct classes of component:

- The Client Applications which make use of the TEE Client API.
- The TEE Client API library implementation.
- The communications stack which is shared amongst all Client Applications, and whose role is to handle the communications between the rich environment and the trusted environment.

As before, there is no mandated architecture for these components and they are only used as logical constructions within this specification document. Note that the TEE Client API may be exposed to either, or both, the privileged or user layers of the rich environment. If exposed in the privileged layer, then drivers or any other privileged components may be considered to take the place of Client Applications.

3. Principles and Concepts

This section explains the underlying principles and concepts of the TEE Client API in detail, explaining how each class of features should be used.

3.1. Design Principles

The key design principles of the TEE Client API are:

- **C language**:
  - C is the common denominator for practically all of the application frameworks and operating systems hosting Client Applications.
  - It is accepted that alternative language bindings – such as a Java API – may be needed in the future, but these are outside of the scope of this specification.
- **Blocking functions**:
  - Most Client Application developers are familiar with synchronous functions which block waiting for the underlying task to complete before returning to the calling code. An asynchronous interface is hard to design, hard to port in rich OS environments, and is generally difficult to use for developers familiar with synchronous APIs.
  - In addition it is assumed that multi-threading support is available on all target platforms; this is required for Implementations to support cancellation of blocking API functions.
- **Source-level portability**:
  - To enable compile-time and design-time optimization, this standard places no requirement on binary compatibility. Client Application developers will need to recompile their code against an appropriate *implementation-defined* version of the TEE Client API headers in order to function correctly on that Implementation.
- **Client-side memory allocations**:
  - Where possible the design of the TEE Client API has placed the responsibility for memory allocation on the calling Client Application code. This gives the Client developer choice of memory allocation locations, enabling simple optimizations such as stack-based allocation or enhanced flexibility using placements in static global memory or thread-local storage.
  - This design choice is evident in the API by the use of pointers to structures rather than opaque handles to represent any manipulated objects.
- **Aim for zero-copy data transfer**:
  - The features of the TEE Client API are chosen to maximize the possibility of zero-copy data transfer between the Client Application and the Trusted Application, provided that the host operating system and hardware implementation can support it. This minimizes communications overhead and improves software efficiency, especially on cached processors where data copies are an expensive operation because of the cache pollution they cause.
  - However, short messages can also be passed by copy, which avoids the overhead of sharing memory.
- **Support memory sharing by pointers**:
  - The TEE Client API will be used to implement higher-level APIs, such as cryptography or secure storage, where the caller will often provide memory buffers for input or output data using simple C pointers. The TEE Client API must allow efficient sharing of this type of memory, and as such does not rely on the Client Application being able to use bulk memory buffers allocated by the TEE Client API.

- **Specify only communication mechanisms**:
    - This API focuses on defining the underlying communications channel. It does not define the format of the messages which pass over the channel, or the protocols used by specific Trusted Applications. These must be defined in other specifications.

3.2. Fundamental Concepts

This section outlines the behavior of the TEE Client API, and introduces key concepts and terminology.

3.2.1. TEE Contexts

A *TEE Context* is an abstraction of the logical connection which exists between a Client Application and a TEE. A TEE Context must be initialized before a Session can be created between the Client Application and a Trusted Application running within the TEE which that TEE Context represents. The TEE Context should be finalized when the connection with the TEE is no longer required, allowing resources to be released.

It is possible for a Client Application to initialize multiple TEE Contexts concurrently, either with the same underlying TEE, or with multiple TEEs if they are available. The number of concurrent contexts which may exist is *implementation-defined*, and may additionally depend on run-time resource constraints.

3.2.2. Sessions

A *Session* is an abstraction of the logical connection which exists between a Client Application and a specific Trusted Application. A Session is opened by the Client Application within the scope of a particular TEE Context. The number of concurrent Sessions which may exist is *implementation-defined*, depending on the design of the TEE and the Trusted Applications in use, and may additionally depend on run-time resource constraints.

When creating a new Session the Client Application must identify the Trusted Applications which it wishes to connect to using the Universally Unique IDentifier (UUID) of the Trusted Application. The open session operation allows an initial data exchange to be made with the Trusted Application, if this is required in the protocol between the Client Application and the Trusted Application.

Connection Methods: Login

Some Trusted Applications may require the Implementation to identify or authenticate the Client Application or the user executing it. For example, a Trusted Application may restrict access to the data or functionality it provides based on the identity of the user running the Client Application in the rich operating environment. When opening a Session the Client Application can nominate which connection method it wants to use and hence which login credentials are presented to the TEE or Trusted Application. It is likely that the connection method will form part of the protocol exposed by the Trusted Application in use; attempting to open a Session with an incorrect connection method may result in a failed attempt.

3.2.3. Commands

A *Command* is the unit of communication between a Client Application and a Trusted Application within a Session. When starting a new Command the Client Application identifies the function in the Trusted Application which it wishes to execute by passing a numeric identifier, and may also provide an operation payload in accordance with the protocol the Trusted Application exposes for that function. The Command invocation blocks the Client Application thread, waiting for an answer from the Trusted Application. A Client Application may use multiple threads to have multiple Commands which are outstanding concurrently. The number of concurrent Commands which may exist is *implementation-defined*, depending on the design of the TEE and the Trusted Applications in use, and may additionally depend on run-time resource constraints.

Operation Payload

An operation to open a Session or to invoke a generic Command can carry an optional payload, the definition of which is passed inside a set of *Operation Parameters* (see section 3.2.5) stored in the operation structure. In this version of the specification up to 4 Parameters can be specified for each operation.

Each Parameter is either a Memory Reference or a Value Parameter and is associated with a direction: it can be input, output, or both input and output. For Memory Reference Parameters, the specified direction of data flow determines when the underlying memory buffers need to be synchronized with the Trusted Application.

Memory Reference Parameters are used to exchange data through shared memory buffers. Value Parameters carry a small amount of data in the form of two 32-bit integers without the burden of sharing or synchronizing memory.

The format of the data structures held in the Memory References or Value Parameters is defined by the protocol of the Trusted Application function in use, and hence outside of the scope of this specification.

Temporary Shared Memory Registration

Memory References refer either to a Registered Memory Reference or a Temporary Memory Reference:

- a *Registered Memory Reference* is a region within a block of *Shared Memory* (see section 3.2.4) that was created before the operation
- a *Temporary Memory Reference* directly specifies a buffer of memory owned by the Client Application, which is temporarily registered by the TEE Client API for the duration of the operation being performed

A Temporary Memory Reference may be null, which can be used to denote a special case for the parameter. Output Memory References that are null are typically used to request the required output size.

Return Codes and Return Origins

The answer to an open Session and a invoke Command operation always contains a *Return code*, which is a 32-bit numeric value indicating success or the reason for failure, and an *Return origin*, which is a 32-bit numeric value indicating the source of the return code in the Implementation. The standard error codes and return origins are described in sections 4.4.2 and 4.4.3.

When the return origin is `TEEC_ORIGIN_TRUSTED_APP` then the return code is defined by the Trusted Application's protocol. Note that, critically, this means that a Client Application cannot just test against `TEEC_SUCCESS`, as the Trusted Application may use another code to indicate success. To enable simpler error handling code in the Client Application it is recommended that the Trusted Application developers choose '0' as their literal value of their success return code constant.

Events and Callbacks

This specification does not define a primitive way for a Trusted Application to spontaneously signal an event to the Client Application or perform callbacks to the Client code. However, these types of usage patterns can be constructed using Commands. For example, event signals can be implemented by having the Client Application send a Command which blocks inside the Trusted Application until the event occurs inside the TEE. When the event occurs the Trusted Application passes control back to the Client Application; the `TEEC_InvokeCommand` will return and the Client Application can handle the event which was signaled.

3.2.4. Shared Memory

A *Shared Memory* block is a region of memory allocated in the context of the Client Application memory space that can be used to transfer data between that Client Application and a Trusted Application.

A Shared Memory block can either be existing Client Application memory which is subsequently registered with the TEE Client API, or memory which is allocated on behalf of the Client Application using the TEE Client API. A Shared Memory block can be registered or allocated once and then used in multiple Commands, and even in multiple Sessions, provided they exist within the scope of the TEE Context in which the Shared Memory was created. This pre-registration is typically more efficient than registering a block of memory using temporary registration if that memory buffer is used in more than one Command invocation.

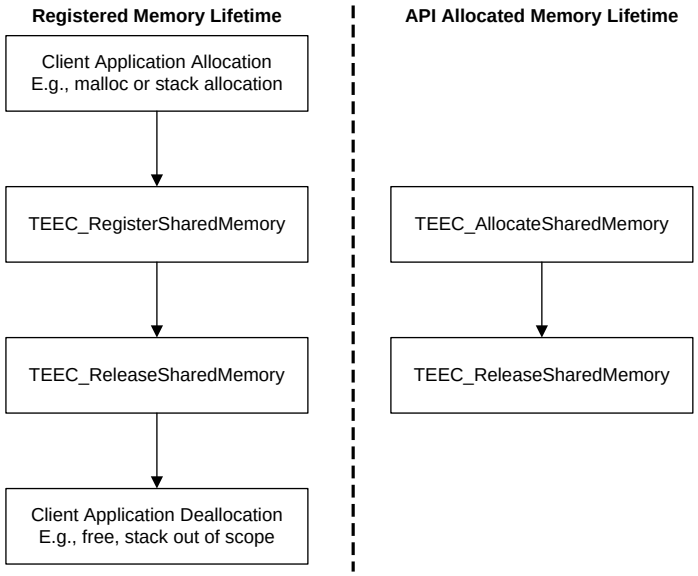

Figure 3-1: Shared Memory Buffer Lifetime

Zero-copy Data Transfer

When possible the implementation of the communications channel beneath the TEE Client API should try to directly map Shared Memory in to the Trusted Application memory space, enabling true zero-copy data transfer. However this is not always possible; for example, the TEE may not have access to the same physical memory system as the platform running the Client Application, or may only be able to achieve zero-copy for some types of memory. As a result this specification defines synchronization points where the TEE Client API Implementation is allowed to synchronize the data in a Shared Memory block with the TEE to ensure data consistency. The Client Application and Trusted Application must assume that the data is only synchronized when within the scope of these synchronization points. Otherwise, data corruption may result. This process is described in more detail in section 3.2.5.

Client Application developers should note that letting the TEE Client API allocate the memory buffers using the function `TEEC_AllocateSharedMemory` maximizes the chances that it can be successfully shared using a zero-copy exchange. If Client Application developers have the option to use this type of allocated memory in their code, without needing an explicit copy from another buffer, then they should aim to do so. However, it is not always possible to allocate memory without a copy in the Client Application, and in these cases registration of the buffer using `TEEC_RegisterSharedMemory` is the preferred option as there is still a possibility that it could be zero copy.

Note that for small amount of data, it is recommended to use a Value Parameter instead of a Memory Reference to avoid the overhead of memory management.

Overlapping Blocks

The API allows Shared Memory registrations and allocations to overlap. A single region of Client Application memory may be registered multiple times, or a block may be allocated and then subsequently registered. The Client is responsible for ensuring that the overlapping regions are consistent and meet any timing requirements when used by multiple actors; specifying an input buffer to one Trusted Application which is concurrently used as an output for another can produce undefined results, for example.

The rules which the Client must conform to when overlapping memory ranges are used concurrently are described in the synchronization sub-section of section 3.2.5.

3.2.5. Memory References

A *Memory Reference* is a range of bytes which is actually shared for a particular operation. A Memory Reference is described by either a `TEEC_MemoryReference` structure (see section 4.3.7) or a `TEEC_TempMemoryReference`. It can specify:

- A whole Shared Memory block.
- A range of bytes within a Shared Memory block.
- Or a pointer to a buffer of memory owned by the Client Application, in which case this buffer is temporarily registered for the duration of the operation. This type of Memory Reference uses the structure `TEEC_TempMemoryReference`.

A Memory Reference also specifies the direction in which data flows for that particular command. Memory References may be marked as input (buffer is transferring data from the Client Application to the Trusted Application), output (buffer is transferring data from the Trusted Application to the Client Application), or both input and output.

When a Memory Reference points to a Shared Memory block, the data flow direction must be consistent with the set of flags defined by the parent Shared Memory block; for example, trying to make an input Memory Reference with a parent Shared Memory block which has only the `TEEC_MEM_OUTPUT` flag is invalid.

Synchronization

As the underlying communications system may not support direct mapping of Client memory into the Trusted Application, it may be necessary to copy a portion of memory from the Client memory space into the Trusted Application memory space. Memory References provide a token which indicates what memory range needs to be synchronized, and their use within an operation indicates the duration of the synchronization scope. The temporal states in this synchronization process are indicated in Figure 3-2.

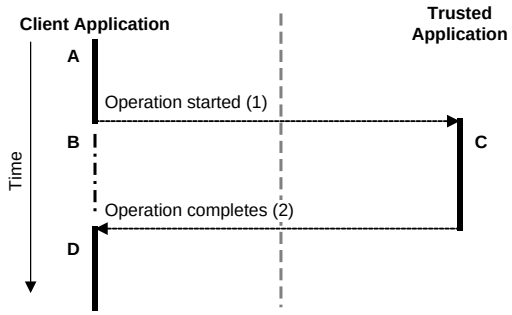

Figure 3-2: Memory Reference timing diagram

In this figure there are three temporal states for the Client application (A, B, D) and one for the Trusted application (C), as well as two synchronization operations (1, 2).

When performing synchronization operation 1 – transitioning from state A to states B and C (which exist in parallel in the two environments) – the Implementation needs to ensure that input buffers are synchronized from the Client Application's view of memory to the Trusted Application's view of it. When performing synchronization operation 2 – transitioning from states B and C (which exist in parallel in the two environments) to state D – the Implementation needs to ensure that output buffers are synchronized from the Trusted Application's view of memory to the Client Application's view of memory.

The range of bytes referenced in a Memory Reference is considered live, for synchronization purposes, the moment that the containing operation structure is passed in to either `TEEC_OpenSession` or `TEEC_InvokeCommand`; this live period corresponds to the temporal states B and C in the figure. A Memory Reference is considered to be no longer live when the called API function returns. While a Memory Reference is live the Client Application and the Trusted Application must obey the following constraints:

1. For ranges within a Memory Reference marked as input only, the Client Application may read from the memory range, but must not write within it (states B and C). The Trusted Application may read from the memory range during state C.
2. For ranges within a Memory Reference marked as input or input and output, the Client Application must neither read nor write within the memory range (state B). The Trusted Application may read and write to the memory range (state C).

If these synchronization rules are ignored by the Client Application or the Trusted Application then data corruption may occur.

Overlapping Ranges

The API allows Memory References to overlap, either within a single operation or across multiple operations. The Client is responsible for ensuring that the overlapping regions are consistent and meet any timing requirements when used by multiple actors; specifying an input buffer to one Trusted Application which is concurrently used as an output for another will produce undefined results, for example.

It may be necessary for constraints on overlapping ranges to be defined as part of the Trusted Application's protocol. A Trusted Application which accepts an input buffer and an output buffer, but which writes to the output buffer before using the input, cannot use the same memory for both activities as writing the output will destroy the input.

Memory Reference Types

The specification supports the following types of Memory Reference which may be encoded in an operation payload.

- `TEEC_MEMREF_TEMP_INPUT`, `TEEC_MEMREF_TEMP_OUTPUT`, or `TEEC_MEMREF_TEMP_INOUT`: A temporary Memory Reference indicates that the Parameter points to a buffer of memory to be shared rather than to a Shared Memory control structure. This Client Application buffer will be temporarily shared for the duration of the operation being performed. If the buffer pointer is `NULL` then no memory buffer is actually referenced. Some Trusted Applications may associate a specific meaning with a null Memory Reference, so for full details the Client Application developer must refer to the protocol specification for the Trusted Application they are targeting. A null Memory Reference can also be used to fetch the required size of an output buffer.

- TEEC_MEMREF_WHOLE: A whole Memory Reference enables a light-weight mechanism of sharing an entire parent Shared Memory block without the need to duplicate the content of the Shared Memory structure control fields inside the Memory Reference. When this memory type is used the entire Shared Memory region is shared with the direction flags the parent Shared Memory specifies.
- TEEC_MEMREF_PARTIAL_INPUT, TEEC_MEMREF_PARTIAL_OUTPUT, or TEEC_MEMREF_PARTIAL_INOUT: A partial Memory Reference refers to a sub-region of a parent Shared Memory block, allowing any region of bytes within that block to be shared with the Trusted Application.

Note that an Operation Parameter can also be a Value Parameter, carrying two 32-bit integers.

Variable Length Return Buffers

In many cases the Trusted Application will want to write a variable length of data in to the Shared Memory buffer. For buffers which are configured as an output buffer, the size of the Memory Reference when starting an Operation on the TEE is the maximum size of the output data that the Trusted Application may write into the referenced region. When the Trusted Application responds it may reduce the size of the referenced memory region to reflect the actual number of bytes it wrote into the output buffer. In this case the Implementation must update the size field of the Memory Reference in the Client Application operation structure to indicate the number of bytes which were used by the Trusted Application.

In these cases the Implementation only needs to synchronize the number of bytes which the Trusted Application has modified when passing control back to the Client Application; other data within the scope of the originally referenced memory range should be unchanged, although this may depend on Trusted Application behaving correctly.

Note that output data can only be written in the lowest address in an output Memory Reference; it is not possible to synchronize a high region in the buffer without synchronizing the lower parts of the buffer.

In any scenario using variable length outputs there is the possibility that the output buffer provided by the Client Application is not large enough to contain the entire output. In these scenarios the Trusted Application is allowed to return the required output size to the Client Application. The size field of the Memory Reference in the operation structure is then updated to reflect the required size, but the Implementation does not synchronize any data with the Client Application, as this is viewed as an error condition. It is recommended that a Trusted Application use the defined “short buffer” error code TEEC_ERROR_SHORT_BUFFER to signal this type of response to the Client Application.

This type of “short buffer” response is allowed for null Memory Reference, enabling a design where a first invocation uses a null Memory Reference to fetch the required size of output buffer, and then uses a second invocation with another non-null Memory Reference containing an output buffer of the necessary size.

3.3. Usage Concepts

The section outlines some of the usage patterns which the design of the TEE Client API makes use of.

3.3.1. Operation Instantiation

To enable reliable multi-threaded implementations of cancellation this specification defines the concept of *Instantiation* – a mechanism which can be used to put TEEC_Operation structures in to a known state. If an Operation may be cancelled by the Client Application then the Client Application must set the started field of the structure to 0 before calling either the TEEC_OpenSession or TEEC_InvokeCommand function. If a Client Application is single threaded, or is multi-threaded but will never cancel the operation by design, then there is no need for the started field to be initialized.

Atomicity of Field Access

To enable multi-threaded TEE Client API implementations to effectively use the `started` field across multiple-threads without the need for OS level locking, the underlying processor architecture must allow atomic operations – such as “test and set”, “swap”, or “exclusive load and store” – to operate on the `started` field. For this reason the `started` field has been chosen to be 32-bits, as this is a commonly supported data size for atomic operations on the processor architectures of interest.

This atomicity requirement typically means that the `started` fields must be naturally aligned (aligned on a 4-byte boundary); otherwise the atomic instructions in the processors will not function correctly. This requirement is automatically met by compliant C code and toolchains, but many toolchains allow extensions to the C language which allow packed and / or unaligned structures. Client Applications must not use these extensions; TEE Client API implementations are allowed to assume the `started` field can be read or written atomically.

3.3.2. Multi-threading

The TEE Client API is designed to support use from multiple threads concurrently, using a combination of internal thread safety within the implementation of the API, and explicit locks and serialization in the Client Application code. Client Application developers can assume that all of the API functions can be used concurrently unless an exception is documented in this specification. The main exceptions are indicated below.

Note that the API can be used from multiple processes, but it may not be possible to share contexts and sessions between multiple processes due to rich OS memory separation mechanisms.

Behavior which is not Thread-safe

TEE Contexts, Sessions, and Shared Memory structures all have an explicit lifecycles defined by pairs of bounding “start” and “stop” functions:

- `TEEC_InitializeContext` / `TEEC_FinalizeContext`
- `TEEC_OpenSession` / `TEEC_CloseSession`
- `TEEC_RegisterSharedMemory` / `TEEC_ReleaseSharedMemory`
- `TEEC_AllocateSharedMemory` / `TEEC_ReleaseSharedMemory`

These functions are not internally thread-safe with respect to the object being initialized or finalized. It is not valid to call `TEEC_OpenSession` concurrently using the same `TEEC_Session` structure, for example. However, it is valid for the Client Application to concurrently use these functions to initialize or finalize different objects; in the above example two threads could initialize different `TEEC_Session` structures.

In cases where global shared structures need to be initialized the Client Application must ensure that the initialization of each structure only occurs once using appropriate platform-specific locking schemes to ensure that this requirement is met.

Once the structures described above have been initialized it becomes possible to use them concurrently in other API functions, provided that the TEE and Trusted Application in use support such concurrent use. A Client Application can concurrently register two different Shared Memory blocks using the same TEE Context, or invoke two Commands within the same Session for example.

3.3.3. Resource Cleanup

The specification of the “stop” functions described in section 3.3.2 is stateful and requires clean Client Application resource unwinding:

- when releasing Shared Memory, the Client code must ensure that it is not referenced in a pending operation

- when closing a session, there must be no pending operations within it
- when finalizing a TEE Context there must be no open sessions within its scope

The Client Applications must ensure these conditions are true, using platform-specific locking mechanisms to synchronize threads if needed. Failing to meet these obligations is a *programmer error*, and will result in undefined behavior.

3.4. Security

This section outlines the security policies of the TEE Client API, and highlights some of the design requirements which are placed on an Implementation.

3.4.1. Security of the TEE and Trusted Applications

The implementation of the TEE and any Trusted Applications must treat any input from the rich environment as potentially malicious; Client Applications are running outside of the TEE security boundary and as such it must be assumed that they may be compromised by attack or may be purposefully malicious.

In particular the following details may be of interest to a TEE or a Trusted Application developer:

- Shared Memory is memory owned by the rich environment and mapped into the TEE memory space. Code inside the TEE and Trusted Applications must assume that the content of Shared Memory is both untrusted and volatile; data stored in Shared Memory may be changed maliciously at any time with respect to the execution of code inside the trusted environment. Note that a well formed Client Application must follow the conventions for sharing memory, as described in section 3.2.5, in order to run with defined behavior.

Login Connection Methods

This specification defines a number of connection methods which allow an identity token for a Client Application to be generated by the Implementation and presented to the Trusted Application. This identity information is generated based on parameters controlled by some trusted entity inside the rich operating system, such as the OS kernel, and as such it is a valid security model for these login tokens to be generated by a trusted process within the rich operating system rather than by the TEE itself. Trusted Application developers must therefore note that the validity of this login token is therefore bounded by the security of the rich operating system, not the security of the TEE.

3.4.2. Security of the Rich Operating System

In most implementations the TEE is a separate operating system which exists in parallel to the rich operating system which runs the Client Applications. It is important that the integration of a TEE alongside the rich operating system cannot be used to weaken the security of the rich operating system itself. The implementation of the TEE Client API, the TEE, and the Trusted Application must ensure that Client Applications cannot use the features they expose to bypass the security sandbox used by the rich operating system to isolate processes.

4. Specification

This section contains the technical specification of the TEE Client API.

4.1. Implementation-Defined Behavior and Programmer Errors

A number of functionalities within this specification are described as either *implementation-defined* or as *programmer errors*.

Implementation-Defined Behavior

When a functional behavior is described as *implementation-defined* it means that a specific Implementation of the TEE Client API MUST consistently implement the behavior and MUST document it. However, the actual behavior is not specified as part of the standard. Client Application developers may choose to depend on this *implementation-defined* behavior, but must be aware that their code may not be portable to another Implementation.

Implementation-Defined Fields

Implementations are allowed to extend some of the data structures defined in this specification to include a single field of *implementation-defined* type, named `imp`. Implementations MUST NOT add new fields outside of `imp`. The implementation can use the `imp` field to hold any private data that it wants to attach to the context which that structure represents, but the Client Application code MUST NOT directly access the contents of the `imp` field.

Programmer Error

There are a number of errors in this specification which can only occur as a result of programmer error, i.e. they are triggered by incorrect use of the API by a program rather than by run-time errors such as out-of-memory conditions. In these cases the Implementation is not required to gracefully handle the error, or even behave consistently, but MAY choose to generate a programmer visible response. This response could include a failing assertion, an informative return code if the function can return one, a diagnostic log file, etc. In these cases the Implementation MUST still guarantee the stability and security of the TEE and the shared communication subsystem in the rich environment because these modules are shared amongst all Client Applications and must not be affected by the misbehavior of a single Client Application.

4.2. Header File

The header file for the TEE Client API must have the name "`tee_client_api.h`".

```
#include "tee_client_api.h"
```

4.3. Data Types

4.3.1. Basic Types

This specification makes use of standard C data types, including the fixed width integer types from the ISO C99 specification update [1]. The following standard C types are used:

- `uint32_t`: a 32-bit unsigned integer
- `uint16_t`: a 16-bit unsigned integer
- `uint8_t`: an 8-bit unsigned integer
- `char`: a character
- `size_t`: an unsigned integer large enough to hold the size of an object in memory

4.3.2. TEEC_Result

This type is used to contain return codes which are the results of invoking TEE Client API functions. See section 4.4.2 for a list of return codes defined by the specification.

```
typedef uint32_t TEEC_Result;
```

4.3.3. TEEC_UUID

This type contains a Universally Unique Resource Identifier (UUID) type as defined in RFC4122 [2]. These UUID values are used to identify Trusted Applications.

```
typedef struct
{
    uint32_t timeLow;
    uint16_t timeMid;
    uint16_t timeHiAndVersion;
    uint8_t  clockSeqAndNode[8];
} TEEC_UUID;
```

4.3.4. TEEC_Context

This type denotes a TEE Context, the main logical container linking a Client Application with a particular TEE. Its content is entirely *implementation-defined*.

```
typedef struct
{
    <Implementation-Defined Type> imp;
} TEEC_Context;
```

4.3.5. TEEC_Session

This type denotes a TEE Session, the logical container linking a Client Application with a particular Trusted Application. Its content is entirely *implementation-defined*.

```
typedef struct
{
    <Implementation-Defined Type> imp;
} TEEC_Session;
```

4.3.6. TEEC_SharedMemory

This type denotes a Shared Memory block which has either been registered with the Implementation or allocated by it.

```
typedef struct
{
    void*    buffer;
    size_t   size;
    uint32_t flags;
    <Implementation-Defined Type> imp;
} TEEC_SharedMemory;
```

The fields of this structure have the following meaning:

- `buffer` is a pointer to the memory buffer shared with the TEE
- `size` is the size of the memory buffer, in bytes
- `flags` is a bit-vector which can contain the following flags:
    - `TEEC_MEM_INPUT`: the memory can be used to transfer data from the Client Application to the TEE

- TEEC_MEM_OUTPUT: the memory can be used to transfer data from the TEE to the Client Application
- All other bits in this field SHOULD be set to zero, and are reserved for future use

- imp contains any additional *implementation-defined* data attached to the Shared Memory structure

4.3.7. TEEC_TempMemoryReference

This type defines a Temporary Memory Reference. It is used as a TEEC_Operation parameter when the corresponding parameter type is one of TEEC_MEMREF_TEMP_INPUT, TEEC_MEMREF_TEMP_OUTPUT, or TEEC_MEMREF_TEMP_INOUT.

```
typedef struct
{
    void*              buffer;
    size_t             size;
} TEEC_TempMemoryReference;
```

The fields of this structure have the following meaning:

- buffer is a pointer to the first byte of a region of memory which needs to be temporarily registered for the duration of the Operation. This field can be NULL to specify a null Memory Reference.
- size is the size of the referenced memory region, in bytes. When the operation completes, and unless the parameter type is TEEC_MEMREF_TEMP_INPUT, the Implementation must update this field to reflect the actual or required size of the output:
  - If the Trusted Application has actually written some data in the output buffer, then the Implementation MUST update the size field with the actual number of bytes written.
  - If the output buffer was not large enough to contain the whole output, or if it is null, the Implementation MUST update the size field with the size of the output buffer requested by the Trusted Application. In this case, no data has been written into the output buffer. See the paragraph “Variable Length Return Buffers” in section 3.2.5 for more details.

4.3.8. TEEC_RegisteredMemoryReference

This type defines a Registered Memory Reference, i.e., that uses a pre-registered or pre-allocated Shared Memory block. It is used as a TEEC_Operation parameter when the corresponding parameter type is one of TEEC_MEMREF_WHOLE, TEEC_MEMREF_PARTIAL_INPUT, TEEC_MEMREF_PARTIAL_OUTPUT, or TEEC_MEMREF_PARTIAL_INOUT.

```
typedef struct
{
    TEEC_SharedMemory* parent;
    size_t             size;
    size_t             offset;
} TEEC_RegisteredMemoryReference;
```

The fields of this structure have the following meaning:

- parent points to a TEEC_SharedMemory structure. The memory reference refers either to the whole Shared Memory or to a partial region within the Shared Memory block, depending of the parameter type. The data flow direction of the memory reference must be consistent with the flags defined in the parent Shared Memory Block. Note that the parent field MUST NOT be NULL. To encode a null Memory Reference, the Client Application must use a Temporary Memory Reference with the buffer field set to NULL.
- size is the size of the referenced memory region, in bytes:

- The Implementation MUST only interpret this field if the Memory Reference type in the operation structure is not `TEEC_MEMREF_WHOLE`. Otherwise, the size is read from the parent Shared Memory structure.
- When an operation completes, and if the Memory Reference is tagged as "output", the Implementation must update this field to reflect the actual or required size of the output. This applies even if the parameter type is `TEEC_MEMREF_WHOLE`:
  - If the Trusted Application has actually written some data in the output buffer, then the Implementation MUST update the `size` field with the actual number of bytes written.
  - If the output buffer was not large enough to contain the whole output, the Implementation MUST update the `size` field with the size of the output buffer requested by the Trusted Application. In this case, no data has been written into the output buffer. See the paragraph "Variable Length Return Buffers" in section 3.2.5 for more details.

- `offset` is the offset, in bytes, of the referenced memory region from the start of the Shared Memory block:
  - The Implementation MUST only interpret this field if the Memory Reference type in the operation structure is not `TEEC_MEMREF_WHOLE`. Otherwise, the Implementation MUST use the base address of the Shared Memory block.

4.3.9. TEEC_Value

This type defines a parameter that is not referencing shared memory, but carries instead small raw data passed by value. It is used as a `TEEC_Operation` parameter when the corresponding parameter type is one of `TEEC_VALUE_INPUT`, `TEEC_VALUE_OUTPUT`, or `TEEC_VALUE_INOUT`.

```
typedef struct
{
   uint32_t a;
   uint32_t b;
} TEEC_Value;
```

The two fields of this structure do not have a particular meaning. It is up to the protocol between the Client Application and the Trusted Application to assign a semantic to those two integers.

4.3.10. TEEC_Parameter

This type defines a Parameter of a `TEEC_Operation`. It can be a Temporary Memory Reference, a Registered Memory Reference, or a Value Parameter.

```
typedef union
{
    TEEC_TempMemoryReference       tmpref;
    TEEC_RegisteredMemoryReference memref;
    TEEC_Value                     value;
} TEEC_Parameter;
```

The field to select in this union depends on the type of the parameter specified in the `paramTypes` field of the `TEEC_Operation` structure:

Parameter Type	Field to use

`TEEC_VALUE_INPUT`
`TEEC_VALUE_OUTPUT`
`TEEC_VALUE_INOUT`
	`value`

Parameter Type	Field to use
TEEC_MEMREF_TEMP_INPUT TEEC_MEMREF_TEMP_OUTPUT TEEC_MEMREF_TEMP_INOUT	tmpref
TEEC_MEMREF_WHOLE TEEC_MEMREF_PARTIAL_INPUT TEEC_MEMREF_PARTIAL_OUTPUT TEEC_MEMREF_PARTIAL_INOUT	memref

4.3.11. TEEC_Operation

This type defines the payload of either an open Session operation or an invoke Command operation. It is also used for cancellation of operations, which may be desirable even if no payload is passed.

```
typedef struct
{
    uint32_t             started;
    uint32_t             paramTypes;
    TEEC_Parameter       params[4];
    <Implementation-Defined Type> imp;
} TEEC_Operation;
```

The fields of this structure have the following meaning:

- `started` is a field which MUST be initialized to zero by the Client Application before each use in an operation if the Client Application may need to cancel the operation about to be performed.
- `paramTypes` encodes the type of each of the Parameters in the operation. The layout of these types within a 32-bit integer is *implementation-defined* and the Client Application MUST use the macro `TEEC_PARAMS_TYPE` to construct a constant value for this field. As a special case, if the Client Application sets `paramTypes` to 0, then the Implementation MUST interpret it as meaning that the type for each Parameter is set to `TEEC_NONE`.

  The type of each Parameter can take one of the following values, which are defined in Table 4-5 (section 4.4.4):
  - `TEEC_NONE`
  - `TEEC_VALUE_INPUT`
  - `TEEC_VALUE_OUTPUT`
  - `TEEC_VALUE_INOUT`
  - `TEEC_MEMREF_TEMP_INPUT`
  - `TEEC_MEMREF_TEMP_OUTPUT`
  - `TEEC_MEMREF_TEMP_INOUT`
  - `TEEC_MEMREF_WHOLE`
  - `TEEC_MEMREF_PARTIAL_INPUT`
  - `TEEC_MEMREF_PARTIAL_OUTPUT`
  - `TEEC_MEMREF_PARTIAL_INOUT`
- `params` is an array of four Parameters. For each parameter, one of the `memref`, `tmpref`, or `value` fields must be used depending on the corresponding parameter type passed in `paramTypes` as described in the specification of `TEEC_Parameter` (section 4.3.9).
- `imp` contains any additional *implementation-defined* data attached to the operation structure.

4.4. Constants

The following constants are defined by this specification:

4.4.1. Configuration Settings

The following utility constant, of type `size_t`, is defined by the specification:

Name	Value	Comment
TEEC_CONFIG_SHAREDMEM_MAX_SIZE	>= 0x80000	The maximum size of a single Shared Memory block, in bytes, of both API allocated and API registered memory. This version of the standard requires that this maximum size is greater than or equal to 512kB. In systems where there is no limit imposed by the Implementation then this definition should be defined to be the size of the address space.

Table 4-1: API Configuration Constants

4.4.2. Return Codes

The following function return codes, of type `TEEC_Result` (see section 4.3.2), are defined by the specification:

Name	Value	Description / Cause
TEEC_SUCCESS	0x00000000	The operation was successful.
TEEC_ERROR_GENERIC	0xFFFF0000	Non-specific cause.
TEEC_ERROR_ACCESS_DENIED	0xFFFF0001	Access privileges are not sufficient.
TEEC_ERROR_CANCEL	0xFFFF0002	The operation was cancelled.
TEEC_ERROR_ACCESS_CONFLICT	0xFFFF0003	Concurrent accesses caused conflict.
TEEC_ERROR_EXCESS_DATA	0xFFFF0004	Too much data for the requested operation was passed.
TEEC_ERROR_BAD_FORMAT	0xFFFF0005	Input data was of invalid format.
TEEC_ERROR_BAD_PARAMETERS	0xFFFF0006	Input parameters were invalid.
TEEC_ERROR_BAD_STATE	0xFFFF0007	Operation is not valid in the current state.
TEEC_ERROR_ITEM_NOT_FOUND	0xFFFF0008	The requested data item is not found.
TEEC_ERROR_NOT_IMPLEMENTED	0xFFFF0009	The requested operation should exist but is not yet implemented.
TEEC_ERROR_NOT_SUPPORTED	0xFFFF000A	The requested operation is valid but is not supported in this Implementation.
TEEC_ERROR_NO_DATA	0xFFFF000B	Expected data was missing.
TEEC_ERROR_OUT_OF_MEMORY	0xFFFF000C	System ran out of resources.

Name	Value	Description / Cause
TEEC_ERROR_BUSY	0xFFFF000D	The system is busy working on something else.
TEEC_ERROR_COMMUNICATION	0xFFFF000E	Communication with a remote party failed.
TEEC_ERROR_SECURITY	0xFFFF000F	A security fault was detected.
TEEC_ERROR_SHORT_BUFFER	0xFFFF0010	The supplied buffer is too short for the generated output.
Implementation-Defined	0x00000001 – 0xFFFEFFFF	
Reserved for Future Use	0xFFFF0011 – 0xFFFFFFFF	

Table 4-2: API Return Code Constants

4.4.3. Return Code Origins

The following function return code origins, of type uint32_t, are defined by the specification. These indicate where in the software stack the return code was generated for an open-session operation or an invoke-command operation.

Name	Value	Comment
TEEC_ORIGIN_API [1]	0x00000001	The return code is an error that originated within the TEE Client API implementation.
TEEC_ORIGIN_COMMS [1]	0x00000002	The return code is an error that originated within the underlying communications stack linking the rich OS with the TEE.
TEEC_ORIGIN_TEE [1]	0x00000003	The return code is an error that originated within the common TEE code.
TEEC_ORIGIN_TRUSTED_APP	0x00000004	The return code originated within the Trusted Application code. This includes the case where the return code is a success.
All other values Reserved for Future Use		

[1] These errors are returned by the TEE or surrounding framework, and the return codes for these error origins must be one of the explicitly-defined constants in Table 4-2. In these cases, the return code cannot be TEEC_SUCCESS because a success can only be generated by the Trusted Application itself.

Table 4-3: API Return Code Origin Constants

4.4.4. Shared Memory Control

The following flag constants, of type uint32_t, are defined by the specification. These are used to indicate the current status and synchronization requirements of Shared Memory blocks.

Name	Value	Comment
TEEC_MEM_INPUT	0x00000001	The Shared Memory can carry data from the Client Application to the Trusted Application.
TEEC_MEM_OUTPUT	0x00000002	The Shared Memory can carry data from the Trusted Application to the Client Application.
All other flag values Reserved for Future Use		

Table 4-4: API Shared Memory Control Flags

The following constants, of type `uint32_t`, are defined by the specification. These are used to indicate the type of Parameter encoded inside the operation structure.

Name	Value	Comment
TEEC_NONE	0x00000000	The Parameter is not used
TEEC_VALUE_INPUT	0x00000001	The Parameter is a `TEEC_Value` tagged as input.
TEEC_VALUE_OUTPUT	0x00000002	The Parameter is a `TEEC_Value` tagged as output.
TEEC_VALUE_INOUT	0x00000003	The Parameter is a `TEEC_Value` tagged as both as input and output, i.e., for which both the behaviors of `TEEC_VALUE_INPUT` and `TEEC_VALUE_OUTPUT` apply.
TEEC_MEMREF_TEMP_INPUT	0x00000005	The Parameter is a `TEEC_TempMemoryReference` describing a region of memory which needs to be temporarily registered for the duration of the Operation and is tagged as input.
TEEC_MEMREF_TEMP_OUTPUT	0x00000006	Same as `TEEC_MEMREF_TEMP_INPUT`, but the Memory Reference is tagged as output. The Implementation may update the `size` field to reflect the required output size in some use cases.
TEEC_MEMREF_TEMP_INOUT	0x00000007	A Temporary Memory Reference tagged as both input and output, i.e., for which both the behaviors of `TEEC_MEMREF_TEMP_INPUT` and `TEEC_MEMREF_TEMP_OUTPUT` apply.
TEEC_MEMREF_WHOLE	0x0000000C	The Parameter is a Registered Memory Reference that refers to the entirety of its parent Shared Memory block. The parameter structure is a `TEEC_MemoryReference`. In this structure, the Implementation MUST read only the `parent` field and MAY update the `size` field when the operation completes.

Name	Value	Comment
TEEC_MEMREF_PARTIAL_INPUT	0x0000000D	A Registered Memory Reference structure that refers to a partial region of its parent Shared Memory block and is tagged as input.
TEEC_MEMREF_PARTIAL_OUTPUT	0x0000000E	A Registered Memory Reference structure that refers to a partial region of its parent Shared Memory block and is tagged as output.
TEEC_MEMREF_PARTIAL_INOUT	0x0000000F	The Registered Memory Reference structure that refers to a partial region of its parent Shared Memory block and is tagged as both input and output, i.e., for which both the behaviors of TEEC_MEMREF_PARTIAL_INPUT and TEEC_MEMREF_PARTIAL_OUTPUT apply.
All other values Reserved for Future Use		

Table 4-5: API Parameter Types

4.4.5. Session Login Methods

The following constants, of type uint32_t, are defined by the specification. These are used to indicate what identity credentials about the Client Application are used by the Implementation to determine access control permissions to functionality provided by, or data stored by, the Trusted Application.

Login types are designed to be orthogonal from each other, in accordance with the identity token(s) defined for each constant. For example, the credentials generated for TEEC_LOGIN_APPLICATION MUST only depend on the identity of the application program, and not the user running it. If two users use the same program, the Implementation MUST assign the same login identity to both users so that they can access the same assets held inside the TEE. These identity tokens MUST also be persistent within one Implementation, across multiple invocations of the application and across power cycles, enabling them to be used to disambiguate persistent storage. Note that this specification does not guarantee separation based on use of different login types – in many embedded platforms there is no notion of "group" or "user" so these login types may fall back to TEEC_LOGIN_PUBLIC – these details of generating the credential for each login type are *implementation-defined*.

Name	Value	Comment
TEEC_LOGIN_PUBLIC	0x00000000	No login data is provided.
TEEC_LOGIN_USER	0x00000001	Login data about the user running the Client Application process is provided.
TEEC_LOGIN_GROUP	0x00000002	Login data about the group running the Client Application process is provided.
TEEC_LOGIN_APPLICATION	0x00000004	Login data about the running Client Application itself is provided.

Name	Value	Comment
TEEC_LOGIN_USER_APPLICATION	0x00000005	Login data about the user running the Client Application and about the Client Application itself is provided.
TEEC_LOGIN_GROUP_APPLICATION	0x00000006	Login data about the group running the Client Application and about the Client Application itself is provided.
Reserved for *implementation-defined* connection methods	0x80000000 – 0xFFFFFFFF	Behavior is *implementation-defined*.
All other constant values Reserved for Future Use		

Table 4-6: API Session Login Methods

4.5. Functions

The following sub-sections specify the behavior of the functions within the TEE Client API.

4.5.1. Documentation Format

```
Function Prototype
```

Description

This section describes the behavior of the function.

Parameters

This section describes each of the function parameters.

Return

This section lists the possible return values. Note that this section is not comprehensive, and often leaves some choice over error returns to the Implementation. However, in cases where restrictions do exist then this section will document them.

Programmer Error

This section documents the cases of *programmer error* – error cases which MAY be detected by the Implementation, but which MAY also perform in an unpredictable manner. This section is not exhaustive, and does not document cases such as passing of an invalid pointer or a `NULL` pointer where the body text states that the pointer must point to a valid structure.

Implementers' Notes

This section highlights key points about the intended use to the implementer.

4.5.2. TEEC_InitializeContext

```
TEEC_Result TEEC_InitializeContext(
    const char*   name,
    TEEC_Context* context)
```

Description

This function initializes a new TEE Context, forming a connection between this Client Application and the TEE identified by the string identifier `name`.

The Client Application MAY pass a `NULL name`, which means that the Implementation MUST select a default TEE to connect to. The supported name strings, the mapping of these names to a specific TEE, and the nature of the default TEE are *implementation-defined*.

The caller MUST pass a pointer to a valid TEEC Context in `context`. The Implementation MUST assume that all fields of the `TEEC_Context` structure are in an *undefined* state.

Parameters

- `name`: a zero-terminated string that describes the TEE to connect to. If this parameter is set to `NULL` the Implementation MUST select a default TEE.
- `context`: a `TEEC_Context` structure that MUST be initialized by the Implementation.

Return

- `TEEC_SUCCESS`: the initialization was successful.
- Another error code from Table 4-2: initialization was not successful.

Programmer Error

The following usage of the API is a *programmer error*:

- Attempting to initialize the same TEE Context structure concurrently from multiple threads. Multi-threaded Client Applications must use platform-provided locking mechanisms to ensure that this case does not occur.

Implementers' Notes

It is valid Client Application behavior to concurrently initialize different TEE Contexts, so the Implementation MUST support this.

4.5.3. TEEC_FinalizeContext

```
void TEEC_FinalizeContext(
    TEEC_Context* context)
```

Description

This function finalizes an initialized TEE Context, closing the connection between the Client Application and the TEE. The Client Application MUST only call this function when all Sessions inside this TEE Context have been closed and all Shared Memory blocks have been released.

The implementation of this function MUST NOT be able to fail: after this function returns the Client Application must be able to consider that the Context has been closed.

The function implementation MUST do nothing if `context` is `NULL`.

Parameters

- `context`: an initialized `TEEC_Context` structure which is to be finalized.

Programmer Error

The following usage of the API is a *programmer error*:

- Calling with a `context` which still has sessions opened.
- Calling with a `context` which contains unreleased Shared Memory blocks.
- Attempting to finalize the same TEE Context structure concurrently from multiple threads.
- Attempting to finalize the same TEE Context structure more than once, without an intervening call to `TEEC_InitalizeContext`.

4.5.4. TEEC_RegisterSharedMemory

```
TEEC_Result TEEC_RegisterSharedMemory(
    TEEC_Context*      context,
    TEEC_SharedMemory* sharedMem)
```

Description

This function registers a block of existing Client Application memory as a block of Shared Memory within the scope of the specified TEE Context, in accordance with the parameters which have been set by the Client Application inside the `sharedMem` structure.

The parameter `context` MUST point to an initialized TEE Context.

The parameter `sharedMem` MUST point to the Shared Memory structure defining the memory region to register. The Client Application MUST have populated the following fields of the Shared Memory structure before calling this function:

- The `buffer` field MUST point to the memory region to be shared, and MUST not be `NULL`.
- The `size` field MUST contain the size of the buffer, in bytes. Zero is a valid length for a buffer.
- The `flags` field indicates the intended directions of data flow between the Client Application and the TEE.
- The Implementation MUST assume that all other fields in the Shared Memory structure have *undefined* content.

An Implementation MAY put a hard limit on the size of a single Shared Memory block, defined by the constant `TEEC_CONFIG_SHAREDMEM_MAX_SIZE`. However note that this function may fail to register a block smaller than this limit due to a low resource condition encountered at run-time.

Parameters

- `context`: a pointer to an initialized TEE Context
- `sharedMem`: a pointer to a Shared Memory structure to register:
    - the `buffer`, `size`, and `flags` fields of the `sharedMem` structure MUST be set in accordance with the specification described above

Return

- `TEEC_SUCCESS`: the registration was successful.
- `TEEC_ERROR_OUT_OF_MEMORY`: the registration could not be completed because of a lack of resources.
- Another error code from Table 4-2: registration was not successful for another reason.

Programmer Error

The following usage of the API is a *programmer error*:

- Calling with a `context` which is not initialized.
- Calling with a `sharedMem` which has not be correctly populated in accordance with the specification.
- Attempting to initialize the same Shared Memory structure concurrently from multiple threads. Multi-threaded Client Applications must use platform-provided locking mechanisms to ensure that this case does not occur.

Implementers' Notes

This design allows a non-`NULL` buffer with a size of 0 bytes to allow trivial integration with any implementations of the C library `malloc`, in which is valid to allocate a zero byte buffer and receive a non-`NULL` pointer which may not be de-referenced in return.

Once successfully registered, the Shared Memory block can be used for efficient data transfers between the Client Application and the Trusted Application. The TEE Client API implementation and the underlying communications infrastructure SHOULD attempt to transfer data in to the TEE without using copies, if this is possible on the underlying implementation, but MUST fall back on data copies if zero-copy cannot be achieved. Client Application developers should be aware that, if the Implementation requires data copies, then Shared Memory registration may allocate a block of memory of the same size as the block being registered.

4.5.5. TEEC_AllocateSharedMemory

```
TEEC_Result TEEC_AllocateSharedMemory(
    TEEC_Context*      context,
    TEEC_SharedMemory* sharedMem)
```

Description

This function allocates a new block of memory as a block of Shared Memory within the scope of the specified TEE Context, in accordance with the parameters which have been set by the Client Application inside the `sharedMem` structure.

The `context` parameter MUST point to an initialized TEE Context.

The `sharedMem` parameter MUST point to the Shared Memory structure defining the region to allocate. Client Applications MUST have populated the following fields of the Shared Memory structure:

- The `size` field MUST contain the desired size of the buffer, in bytes. The size is allowed to be zero. In this case memory is allocated and the pointer written in to the `buffer` field on return MUST not be `NULL` but MUST never be de-referenced by the Client Application. In this case however, the Shared Memory block can be used in Registered Memory References.
- The `flags` field indicates the allowed directions of data flow between the Client Application and the TEE.
- The Implementation MUST assume that all other fields in the Shared Memory structure have *undefined* content.

An Implementation MAY put a hard limit on the size of a single Shared Memory block, defined by the constant `TEEC_CONFIG_SHAREDMEM_MAX_SIZE`. However it must be noted that this function may fail to allocate a block of smaller than this limit due to low resource scenarios encountered at run-time.

If this function returns any code other than `TEEC_SUCCESS` the Implementation MUST have set the `buffer` field of `sharedMem` to `NULL`.

Parameters

- `context`: a pointer to an initialized TEE Context.
- `sharedMem`: a pointer to a Shared Memory structure to allocate:
  - Before calling this function, the Client Application MUST have set the `size`, and `flags` fields.
  - On return, for a successful allocation the Implementation MUST have set the pointer `buffer` to the address of the allocated block, otherwise it MUST set `buffer` to `NULL`.

Return

- `TEEC_SUCCESS`: the allocation was successful.
- `TEEC_ERROR_OUT_OF_MEMORY`: the allocation could not be completed due to resource constraints.
- Another error code from Table 4-2: allocation was not successful for another reason.

Programmer Error

The following usage of the API is a *programmer error*:

- Calling with a `context` which is not initialized.
- Calling with `sharedMem` which has not been populated in accordance with the specification.

- Attempting to initialize the same Shared Memory structure concurrently from multiple threads. Multi-threaded Client Applications must use platform-provided locking mechanisms to ensure that this case does not occur.

Implementers' Notes

Once successfully allocated the Shared Memory block can be used for efficient data transfers between the Client Application and the Trusted Application. The TEE Client API and the underlying communications infrastructure should attempt to transfer data in to the TEE without using copies, if this is possible on the underlying implementation, but may have to fall back on data copies if zero-copy cannot be achieved.

The memory buffer allocated by this function must have sufficient alignment to store any fundamental C data type at a natural alignment. For most platforms this will require the memory buffer to have 8-byte alignment, but refer to the Application Binary Interface (ABI) of the target platform for details.

4.5.6. TEEC_ReleaseSharedMemory

```
void TEEC_ReleaseSharedMemory (
    TEEC_SharedMemory* sharedMem)
```

Description

This function deregisters or deallocates a previously initialized block of Shared Memory.

For a memory buffer allocated using `TEEC_AllocateSharedMemory` the Implementation MUST free the underlying memory and the Client Application MUST NOT access this region after this function has been called. In this case the Implementation MUST set the `buffer` and `size` fields of the `sharedMem` structure to `NULL` and `0` respectively before returning.

For memory registered using `TEEC_RegisterSharedMemory` the Implementation MUST deregister the underlying memory from the TEE, but the memory region will stay available to the Client Application for other purposes as the memory is owned by it.

The Implementation MUST do nothing if the `sharedMem` parameter is `NULL`.

Parameters

- `sharedMem`: a pointer to a valid Shared Memory structure.

Programmer Error

The following usage of the API is a *programmer error*:

- Attempting to release Shared Memory which is used by a pending operation.
- Attempting to release the same Shared Memory structure concurrently from multiple threads. Multi-threaded Client Applications must use platform-provided locking mechanisms to ensure that this case does not occur.

4.5.7. TEEC_OpenSession

```
TEEC_Result TEEC_OpenSession (
    TEEC_Context*    context,
    TEEC_Session*    session,
    const TEEC_UUID* destination,
    uint32_t         connectionMethod,
    const void*      connectionData,
    TEEC_Operation*  operation,
    uint32_t*        returnOrigin)
```

Description

This function opens a new Session between the Client Application and the specified Trusted Application.

The Implementation MUST assume that all fields of this `session` structure are in an *undefined* state. When this function returns `TEEC_SUCCESS` the Implementation MUST have populated this structure with any information necessary for subsequent operations within the Session.

The target Trusted Application is identified by a UUID passed in the parameter `destination`.

The Session MAY be opened using a specific connection method that can carry additional connection data, such as data about the user or user-group running the Client Application, or about the Client Application itself. This allows the Trusted Application to implement access control methods which separate functionality or data accesses for different actors in the rich environment outside of the TEE. Standard connection methods are defined in section 4.4.5, but there MAY be *implementation-defined* login methods in addition to these core types. The additional data associated with each connection method is passed in via the pointer `connectionData`. For the core login types the following connection data is required:

- `TEEC_LOGIN_PUBLIC`
  - `connectionData` SHOULD be `NULL`.
- `TEEC_LOGIN_USER`
  - `connectionData` SHOULD be `NULL`.
- `TEEC_LOGIN_GROUP`
  - `connectionData` MUST point to a `uint32_t` which contains the group which this Client Application wants to connect as. The Implementation is responsible for securely ensuring that the Client Application instance is actually a member of this group.
- `TEEC_LOGIN_APPLICATION`
  - `connectionData` SHOULD be `NULL`.
- `TEEC_LOGIN_USER_APPLICATION`
  - `connectionData` SHOULD be `NULL`.
- `TEEC_LOGIN_GROUP_APPLICATION`
  - `connectionData` MUST point to a `uint32_t` which contains the group which this Client Application wants to connect as. The Implementation is responsible for securely ensuring that the Client Application instance is actually a member of this group.

Note: This API intentionally omits any form of support for static login credentials, such as PIN or password entry. The login methods supported in the API are only those which have been identified as requiring support by the rich operating environment. If a Trusted Application requires a static login credential then this can be passed by the Client Application using the standard Shared Memory mechanisms for data exchange.

An open-session operation MAY optionally carry an Operation Payload, and MAY also be cancellable. When the payload is present the parameter `operation` MUST point to a `TEEC_Operation` structure populated by the Client Application. If `operation` is `NULL` then no data buffers are exchanged with the Trusted Application, and the operation cannot be cancelled by the Client Application. The full behavior of the Operation Payload handling is described in section 4.5.9, `TEEC_InvokeCommand`.

The result of this function is returned both in the function `TEEC_Result` return code and the return origin, stored in the variable pointed to by `returnOrigin`:

- If the return origin is different from `TEEC_ORIGIN_TRUSTED_APP`, then the return code MUST be one of the error codes defined in Table 4-2. If the return code is `TEEC_ERROR_CANCEL` then it means that the operation was cancelled before it reached the Trusted Application.
- If the return origin is `TEEC_ORIGIN_TRUSTED_APP`, the meaning of the return code depends on the protocol between the Client Application and the Trusted Application. However, if `TEEC_SUCCESS` is returned, it always means that the session was successfully opened and if the function returns a code different from `TEEC_SUCCESS`, it means that the session opening failed.

Parameters

- `context`: a pointer to an initialized TEE Context.
- `session`: a pointer to a Session structure to open.
- `destination`: a pointer to a structure containing the UUID of the destination Trusted Application.
- `connectionMethod`: the method of connection to use. Refer to section 4.4.5 for more details.
- `connectionData`: any necessary data required to support the connection method chosen.
- `operation`: a pointer to an Operation containing a set of Parameters to exchange with the Trusted Application, or `NULL` if no Parameters are to be exchanged or if the operation cannot be cancelled. Refer to `TEEC_InvokeCommand` for more details.
- `returnOrigin`: a pointer to a variable which will contain the return origin. This field may be `NULL` if the return origin is not needed.

Return

- If the `returnOrigin` is different from `TEEC_ORIGIN_TRUSTED_APP`, an error code from Table 4-2
- If the `returnOrigin` is equal to `TEEC_ORIGIN_TRUSTED_APP`, a return code defined by the protocol between the Client Application and the Trusted Application. In any case, a return code set to `TEEC_SUCCESS` means that the session was successfully opened and a return code different from `TEEC_SUCCESS` means that the session opening failed.

Programmer Error

The following usage of the API is a *programmer error*:

- Calling with a `context` which is not yet initialized.
- Calling with a `connectionData` set to `NULL` if connection data is required by the specified connection method.
- Calling with an `operation` containing an invalid `paramTypes` field, i.e., containing a reserved parameter type or where a parameter type that conflicts with the parent Shared Memory .
- Encoding Registered Memory References which refer to Shared Memory blocks allocated within the scope of a different TEE Context.
- Attempting to open a Session using the same Session structure concurrently from multiple threads. Multi-threaded Client Applications must use platform-provided locking mechanisms, to ensure that this case does not occur.
- Using the same Operation structure for multiple concurrent operations.

Implementers' Notes

Trusted Applications MUST use `TEEC_SUCCESS` (0) to indicate success in their protocol, as this is the only way for the Implementation to determine success or failure without knowing the protocol of the Trusted Application.

4.5.8. TEEC_CloseSession

```
void TEEC_CloseSession (
    TEEC_Session* session)
```

Description

This function closes a Session which has been opened with a Trusted Application.

All Commands within the Session MUST have completed before calling this function.

The Implementation MUST do nothing if the `session` parameter is `NULL`.

The implementation of this function MUST NOT be able to fail: after this function returns the Client Application must be able to consider that the Session has been closed.

Parameters

- `session`: the session to close.

Programmer Error

The following usage of the API is a *programmer error*:

- Calling with a `session` which still has commands running.
- Attempting to close the same Session concurrently from multiple threads.
- Attempting to close the same Session more than once.

4.5.9. TEEC_InvokeCommand

```
TEEC_Result TEEC_InvokeCommand(
    TEEC_Session*     session,
    uint32_t          commandID,
    TEEC_Operation*   operation,
    uint32_t*         returnOrigin)
```

Description

This function invokes a Command within the specified Session.

The parameter `session` `MUST` point to a valid open Session.

The parameter `commandID` is an identifier that is used to indicate which of the exposed Trusted Application functions should be invoked. The supported command identifiers are defined by the Trusted Application's protocol.

Operation Handling

A Command MAY optionally carry an Operation Payload. When the payload is present the parameter `operation` MUST point to a `TEEC_Operation` structure populated by the Client Application. If `operation` is `NULL` then no parameters are exchanged with the Trusted Application, and only the Command ID is exchanged.

The `operation` structure is also used to manage cancellation of the Command. If cancellation is required then the `operation` pointer MUST be non-`NULL` and the Client Application MUST have zeroed the `started` field of the operation structure before calling this function. The operation structure MAY contain no Parameters if no data payload is to be exchanged.

The Operation Payload is handled as described by the following steps, which are executed sequentially:

1. Each Parameter in the Operation Payload is examined. If the parameter is a Temporary Memory Reference, then it is registered for the duration of the Operation in accordance with the fields set in the `TEEC_TempMemoryReference` structure and the data flow direction specified in the parameter type. Refer to the `TEEC_RegisterSharedMemory` function for error conditions which can be triggered during temporary registration of a memory region.

2. The contents of all the Memory Regions which are exchanged with the TEE are synchronized (see section 3.2.5).

3. The fields of all Value Parameters tagged as input are read by the Implementation. This applies to Parameters of type `TEEC_VALUE_INPUT` or `TEEC_VALUE_INOUT`.

4. The Operation is issued to the Trusted Application. During the execution of the Command, the Trusted Application may read the data held within the memory referred to by input Memory References. It may also write data in to the memory referred to by output Memory References, but these modifications are not guaranteed to be observable by the Client Application until the command completes.

5. After the Command has completed, the Implementation MUST update the `size` field of the Memory Reference structures flagged as output:

a. For Memory References that are non-null and marked as output, the updated `size` field MAY be less than or equal to original `size` field. In this case this indicates the number of bytes actually written by the Trusted Application, and the Implementation MUST synchronize this region with the Client Application memory space.
b. For all Memory References marked as output, the updated `size` field MAY be larger than the original `size` field. For null Memory References, a required buffer size MAY be specified by the Trusted Application. In these cases the passed output buffer was too small or absent, and the returned size indicates the size of the output buffer which is necessary for the operation to succeed. In these cases the Implementation SHOULD NOT synchronize any shared data with the Client Application.
This behavior is described in more detail in section 3.2.5, “Variable Length Return Buffers”.

6. When the Command completes, the Implementation MUST update the fields of all Value Parameters tagged as output, i.e., of type `TEEC_VALUE_OUTPUT` or `TEEC_VALUE_INOUT`.

7. All memory regions that were temporarily registered at the beginning of the function are deregistered as if the function `TEEC_ReleaseSharedMemory` was called on each of them.

8. Control is passed back to the calling Client Application code

The result of this function is returned both in the function `TEEC_Result` return code and the return origin, stored in the variable pointed to by `returnOrigin`:

- If the return origin is different from `TEEC_ORIGIN_TRUSTED_APP`, then the return code MUST be one of the error codes defined in Table 4-2. If the return code is `TEEC_ERROR_CANCEL` then it means that the operation was cancelled before it reached the Trusted Application.
- If the return origin is `TEEC_ORIGIN_TRUSTED_APP`, then the meaning of the return code is determined by the protocol exposed by the Trusted Application. It is recommended that the Trusted Application developer chooses `TEEC_SUCCESS` (0) to indicate success in their protocol, as this means that it is possible for the Client Application developer to determine success or failure without looking at the return origin.

Parameters

- `session`: the open Session in which the command will be invoked.
- `commandID`: the identifier of the Command within the Trusted Application to invoke. The meaning of each Command Identifier must be defined in the protocol exposed by the Trusted Application
- `operation`: a pointer to a Client Application initialized `TEEC_Operation` structure, or `NULL` if there is no payload to send or if the Command does not need to support cancellation.
- `returnOrigin`: a pointer to a variable which will contain the return origin. This field may be `NULL` if the return origin is not needed.

Return

- if the return origin is different from `TEEC_ORIGIN_TRUSTED_APP`, an error code defined in Table 4-2
- if the return origin is `TEEC_ORIGIN_TRUSTED_APP`, a return code defined by the Trusted Application protocol

Programmer Error

The following usage of the API is a *programmer error*:

- Calling with a `session` which is not an open Session.

- Calling with invalid content in the `paramTypes` field of the `operation` structure. This invalid behavior includes types which are RFU or which conflict with the flags of the parent Shared Memory block.
- Encoding Registered Memory References which refer to Shared Memory blocks allocated or registered within the scope of a different TEE Context.
- Using the same operation structure concurrently for multiple operations, whether open Session operations or Command invocations.

4.5.10. TEEC_RequestCancellation

```
void TEEC_RequestCancellation(
    TEEC_Operation* operation)
```

Description

This function requests the cancellation of a pending open Session operation or a Command invocation operation. As this is a synchronous API, this function must be called from a thread other than the one executing the `TEEC_OpenSession` or `TEEC_InvokeCommand` function.

This function just sends a cancellation signal to the TEE and returns immediately; the operation is not guaranteed to have been cancelled when this function returns. In addition, the cancellation request is just a hint; the TEE or the Trusted Application MAY ignore the cancellation request.

It is valid to call this function using a `TEEC_Operation` structure any time after the Client Application has set the `started` field of an Operation structure to zero. In particular, an operation can be cancelled before it is actually invoked, during invocation, and after invocation. Note that the Client Application MUST reset the `started` field to zero each time an Operation structure is used or re-used to open a Session or invoke a Command if the new operation is to be cancellable.

Client Applications MUST NOT reuse the Operation structure for another Operation until the cancelled command has actually returned in the thread executing the `TEEC_OpenSession` or `TEEC_InvokeCommand` function.

Detecting cancellation

In many use cases it will be necessary for the Client Application to detect whether the operation was actually cancelled, or whether it completed normally.

In some implementations it MAY be possible for part of the infrastructure to cancel the operation before it reaches the Trusted Application. In these cases the return origin returned by `TEEC_OpenSession` or `TEEC_InvokeCommand` MUST be either `TEEC_ORIGIN_API`, `TEEC_ORIGIN_COMMS`, or `TEEC_ORIGIN_TEE`, and the return code MUST be `TEEC_ERROR_CANCEL`.

If the cancellation request is handled by the Trusted Application itself then the return origin returned by `TEEC_OpenSession` or `TEEC_InvokeCommand` MUST be `TEE_ORIGIN_TRUSTED_APP`, and the return code is defined by the Trusted Application's protocol. If possible, Trusted Applications SHOULD use `TEEC_ERROR_CANCEL` for their return code, but it is accepted that this is not always possible due to conflicts with existing return code definitions in other standards.

Parameters

- `operation`: a pointer to a Client Application instantiated Operation structure.

4.5.11. Function-Like Macro: TEEC_PARAM_TYPES

```
uint32_t TEEC_PARAM_TYPES( param0Type, param1Type, param2Type, param3Type)
```

Description

This function-like macro builds a constant containing four Parameter types for use in the `paramTypes` field of a `TEEC_Operation` structure. It accepts four parameters which MUST be taken from the constant values described in Table 4-5.

Note that the way in which the parameter types are packed in a 32-bit integer is *implementation-defined* and a Client MUST use this macro to build the content of the `paramTypes` field. However, the value 0 MUST always be equivalent to all types set to `TEEC_NONE`.

5. Sample Code

This section of the specification walks through some example code demonstrating the fundamental usage principles of the TEE Client API.

5.1. Example Trusted Application Protocol

The Trusted Application used for this example implements some simple cryptographic commands, exposing the protocol described in this section to the Client Application. For the purposes of this example it is assumed that each Session opened between the Client Application and the cryptographic Trusted Application only supports a single concurrent encryption operation and a single concurrent digest operation.

Note: This set of Commands is not meant to represent a real-life protocol, but it is designed to show the various ways that the TEE Client API can be used to communicate with a Trusted Application.

5.1.1. Login Support

The Trusted Application supports the `TEEC_LOGIN_USER` connection method, allowing keys belonging to different users to be stored in an isolated fashion.

5.1.2. Standard Return Codes

All Commands exposed by the Trusted Application return standard `TEEC_SUCCESS` or `TEEC_ERROR_*` constants, enabling simple error handling as this removes the need to check that the return origin is `TEEC_ORIGIN_TRUSTED_APP`.

5.1.3. Encryption Commands

CMD_ENCRYPT_INIT

This function initializes an encryption operation using a previously installed key, identified by the Key ID encoded in the input data. The following Memory References are used in the operation payload:

- `commandID`: 1
- `params[0]`
  - Value: a=Encryption Key ID
  - Data flow direction: input
- `params[1]`
  - Encryption Initialization Vector: Array of 16-bytes, aligned on 1 byte boundary.
  - Data flow direction: input

CMD_ENCRYPT_UPDATE

This function encrypts a buffer of data using the previous encryption state in this session if any chaining methods are in use. The encrypted data is returned to the caller. The following Parameters are used in the operation payload:

- `commandID`: 2
- `params[0]`
  - Input data buffer: a multiple of 16-bytes, aligned on a byte boundary.
  - Data flow direction: input
- `params[1]`
  - Output data buffer: same length as input buffer, aligned on a byte boundary.

- Data flow direction: output

CMD_ENCRYPT_FINAL

This function completes an encryption operation which is no longer needed, releasing resources held inside the Trusted Application:

- `commandID`: 3
- No Parameters are used for this command.

5.1.4. Digest Commands

CMD_DIGEST_INIT

This function initializes a digest operation:

- `commandID`: 4
- No Parameters are used for this command.

CMD_DIGEST_UPDATE

This function adds more data to an in-progress digest. The following Parameters are used in the operation payload:

- `commandID`: 5
- `params[0]`
  - Input data buffer: a buffer of bytes, aligned on a byte boundary.
  - Data flow direction: input

CMD_DIGEST_FINAL

This function completes an in-progress digest. The following Parameters are used in the operation payload:

- `commandID`: 6
- `params[1]`
  - Output data buffer: a buffer of 20 bytes, aligned on a byte boundary.
  - Data flow direction: output

5.2. Example 1: Using the TEE Client API

This example Client Application implements some library code which encrypts an input buffer, returning the encrypted buffer and the digest of the encrypted buffer to the caller.

The prototype of this library function is outlined below:

```
TEEC_Result libraryFunction(
    uint8_t const * inputBuffer,
    uint32_t        inputSize,
    uint8_t*        outputBuffer,
    uint32_t        outputSize,
    uint8_t*        digestBuffer )
```

Full example code for this example, including line numbers, can be found in section 6.

5.2.1. Initializing resources

The first task undertaken by the library function is to allocate the structures needed for the function.

```
/* Allocate TEE Client structures on the stack. */
TEEC_Context      context;
TEEC_Session      session;
TEEC_Operation    operation;

TEEC_Result       result;

TEEC_SharedMemory commsSM;
TEEC_SharedMemory inputSM;
TEEC_SharedMemory outputSM;
```

5.2.2. Connecting to the desired Trusted Application

The next task of the library function is to initiate a connection with the Trusted Application. It first initializes a TEE Context, and then opens a session.

```
/* ==========================================================================
[1] Connect to TEE
========================================================================== */
result = TEEC_InitializeContext(
             NULL,
             &context);
if (result != TEEC_SUCCESS)
{
    goto cleanup1;
}

/* ==========================================================================
[2] Open session with TEE application
========================================================================== */
/* Open a Session with the TEE application. */
result = TEEC_OpenSession(
             &context,
             &session,
             &cryptoTEEApp,
             TEEC_LOGIN_USER,
             NULL,     /* No connection data needed for TEEC_LOGIN_USER. */
             NULL,     /* No payload, and do not want cancellation. */
             NULL);
if (result != TEEC_SUCCESS)
{
    goto cleanup2;
}
```

5.2.3. Allocating communications channel Shared Memory

The next task of the library function is to allocate a block of Shared Memory which can be used for exchanging command and control data, such as initialization vectors for the encryption operation, key ID parameters, and for retrieving the digest result. The code uses the function `TEEC_AllocateSharedMemory` to maximize the chance that the buffer can be directly mapped and to share it only once for multiple commands.

The memory buffer will be used in the following ways for the operations invoked by this library:

- Encrypt Initialize:

- `commsSM.buffer[0x0000-0x000F]`: IV (input)

- Digest Finalize
  - `commsSM.buffer[0x0000-0x0013]`: Digest (output)

```
/* ============================================================================
[3] Initialize the Shared Memory buffers
============================================================================ */
/* [a] Communications buffer. */
commsSM.size  = 20;
commsSM.flags = TEEC_MEM_INPUT | TEEC_MEM_OUTPUT;

/* Use TEE Client API to allocate the underlying memory buffer. */
result = TEEC_AllocateSharedMemory(
             &context,
             &commsSM);
if (result != TEEC_SUCCESS)
{
    goto cleanup3;
}
```

5.2.4. Registering bulk buffers as Shared Memory

The next task of the library function is to register as Shared Memory the bulk input and output buffers provided to us as parameters by the calling code. These buffers could be relatively large, so we register the existing memory with the API rather than allocating memory and then copying data in to / out of it.

Note that the digest operation uses the encryption output buffer as an input, so this is marked with both `TEEC_MEM_INPUT` and `TEEC_MEM_OUTPUT`.

```
/* [b] Bulk input buffer. */
inputSM.size  = inputSize;
inputSM.flags = TEEC_MEM_INPUT;

/* Use TEE Client API to register the underlying memory buffer. */
inputSM.buffer = (uint8_t*)inputBuffer;

result = TEEC_RegisterSharedMemory(
             &context,
             &inputSM);
if (result != TEEC_SUCCESS)
{
    goto cleanup4;
}

/* [c] Bulk output buffer (also input for digest). */
outputSM.size  = outputSize;
outputSM.flags = TEEC_MEM_INPUT | TEEC_MEM_OUTPUT;
outputSM.buffer = outputBuffer;

/* Use TEE Client API to register the underlying memory buffer. */
result = TEEC_RegisterSharedMemory(
             &context,
             &outputSM);
if (result != TEEC_SUCCESS)
{
    goto cleanup5;
}
```

5.2.5. Initialize operations

The next task of the library function is to invoke the two cryptographic library initialize commands within the Trusted Application. The code first populates the operation payload structure for each operation and then invokes it.

```
/* ============================================================================
[4] Perform cryptographic operation initialization commands
============================================================================ */
/* [a] Start the encrypt operation within the TEE application. */
operation.paramTypes = TEEC_PARAM_TYPES(
    TEEC_VALUE_INPUT,
    TEEC_MEMREF_PARTIAL_INPUT,
    TEEC_NONE,
    TEEC_NONE);

/* Write key ID (example uses key ID = 1) in the parameter #0 */
operation.params[0].value.a = 1;

operation.params[1].memref.parent = &commsSM;
operation.params[1].memref.offset = 0;
operation.params[1].memref.size   = 16;

/* Write IV (example uses an IV of all zeros) in to Memory buffer. */
ivPtr = (uint8_t*)commsSM.buffer;
memset(ivPtr, 0, 16);

/* Start the encrypt operation within the TEE application. */
result = TEEC_InvokeCommand(
             &session,
             CMD_ENCRYPT_INIT,
             &operation,
             NULL);
if (result != TEEC_SUCCESS)
{
    goto cleanup6;
}

/* [b] Start the digest operation within the TEE application. */
result = TEEC_InvokeCommand(
             &session,
             CMD_DIGEST_INIT,
             NULL,
             NULL);
if (result != TEEC_SUCCESS)
{
    goto cleanup6;
}
```

5.2.6. Perform cryptographic operations

The next task of the library function is to actually perform the encryption operation and the digest operation on the encrypted result.

```
/* ============================================================================
[5] Perform the cryptographic update commands
============================================================================ */
/* [a] Start the encrypt operation within the TEE application. */
operation.paramTypes = TEEC_PARAM_TYPES(
```

```
        TEEC_MEMREF_WHOLE,
        TEEC_MEMREF_PARTIAL_OUTPUT,
        TEEC_NONE,
        TEEC_NONE);

    operation.params[0].memref.parent = &inputSM;
    /* Note that the other fields of operation.params[0].memref need not be
       initialized because the parameter type is TEEC_MEMREF_WHOLE */

    operation.params[1].memref.parent = &outputSM;
    operation.params[1].memref.offset = 0;
    operation.params[1].memref.size   = outputSize;

    /* Start the encrypt operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_ENCRYPT_UPDATE,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* [b] Start the digest operation within the TEE application. */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_MEMREF_PARTIAL_INPUT,
        TEEC_NONE,
        TEEC_NONE,
        TEEC_NONE);

    /* Note: we use the updated size in the MemRef output by the encryption. */
    operation.params[0].memref.parent = &outputSM;
    operation.params[0].memref.offset = 0;
    operation.params[0].memref.size   = operation.params[1].memref.size;

    /* Start the digest operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_DIGEST_UPDATE,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }
```

5.2.7. Finalizing the commands

The next task of the library function is to finalize the cryptographic commands, which releases the resources held by the encryption operation inside the Trusted Application, and fetches the digest result.

```
    /* ============================================================================
    [6] Perform the cryptographic finalize commands
    ============================================================================ */
    /* [a] Finalize the encrypt operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_ENCRYPT_FINAL,
                NULL,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* [b] Finalize the digest operation within the TEE application. */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_MEMREF_PARTIAL_OUTPUT,
        TEEC_NONE,
        TEEC_NONE,
        TEEC_NONE);

    operation.params[0].parent = &commsSM;
    operation.params[0].offset = 0;
    operation.params[0].size   = 20;

    result = TEEC_InvokeCommand(
                &session,
                CMD_DIGEST_FINAL,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* Transfer digest in to user buffer. */
    memcpy(digestBuffer, commsSM.buffer, 20);
```

5.2.8. Cleaning up

By this point in the code all activity with the Trusted Application has completed, and we can release any resources which we hold. The following command sequence is used to cleanup for this function, unwinding the resource allocations in the order in which they occurred.

```
    /* ============================================================================
    [7] Tidyup resources
    ============================================================================ */
cleanup6:
    TEEC_ReleaseSharedMemory(&outputSM);
cleanup5:
    TEEC_ReleaseSharedMemory(&inputSM);
cleanup4:
    TEEC_ReleaseSharedMemory(&commsSM);
cleanup3:
    TEEC_CloseSession(&session);
cleanup2:
    TEEC_FinalizeContext(&context);
```

6. Appendix: Example Source Code

This section contains the full example code of the example described in the walkthrough found in section 5.2.

```
#include "tee_client_api.h"
#include <string.h>

/* ========================================================================
Store the TEE application UUID in non-volatile memory (code ROM).
======================================================================== */
static const TEEC_UUID cryptoTEEApp =
{
    0x3E93632E, 0xA710, 0x469E,
    { 0xAC, 0xC8, 0x5E, 0xDF, 0x8C, 0x85, 0x90, 0xE1 }
};

/* ========================================================================
Definitions of the Trusted Application Command IDs
======================================================================== */
#define CMD_ENCRYPT_INIT   1
#define CMD_ENCRYPT_UPDATE 2
#define CMD_ENCRYPT_FINAL  3
#define CMD_DIGEST_INIT    4
#define CMD_DIGEST_UPDATE  5
#define CMD_DIGEST_FINAL   6

/* ========================================================================
Implement our library function where the buffers of memory are pre-allocated
by the calling entity. This is a common paradigm when interfacing with other
libraries provided by other providers.
======================================================================== */
TEEC_Result libraryFunction(
    uint8_t const * inputBuffer,
    uint32_t        inputSize,
    uint8_t*        outputBuffer,
    uint32_t        outputSize,
    uint8_t*        digestBuffer
)
{
    /* Allocate TEE Client structures on the stack. */
    TEEC_Context      context;
    TEEC_Session      session;
    TEEC_Operation    operation;

    TEEC_Result       result;

    TEEC_SharedMemory commsSM;
    TEEC_SharedMemory inputSM;
    TEEC_SharedMemory outputSM;

    uint8_t*          ivPtr;
```

```
    /* ==========================================================================
    [1] Connect to TEE
    ========================================================================== */
    result = TEEC_InitializeContext(
                NULL,
                &context);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup1;
    }

    /* ==========================================================================
    [2] Open session with TEE application
    ========================================================================== */
    /* Open a Session with the TEE application. */
    result = TEEC_OpenSession(
                &context,
                &session,
                &cryptoTEEApp,
                TEEC_LOGIN_USER,
                NULL,     /* No connection data needed for TEEC_LOGIN_USER. */
                NULL,     /* No payload, and do not want cancellation. */
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup2;
    }

    /* ==========================================================================
    [3] Initialize the Shared Memory buffers
    ========================================================================== */
    /* [a] Communications buffer. */
    commsSM.size  = 20;
    commsSM.flags = TEEC_MEM_INPUT | TEEC_MEM_OUTPUT;

    /* Use TEE Client API to allocate the underlying memory buffer. */
    result = TEEC_AllocateSharedMemory(
                &context,
                &commsSM);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup3;
    }

    /* [b] Bulk input buffer. */
    inputSM.size  = inputSize;
    inputSM.flags = TEEC_MEM_INPUT;

    /* Use TEE Client API to register the underlying memory buffer. */
    inputSM.buffer = (uint8_t*)inputBuffer;

    result = TEEC_RegisterSharedMemory(
                &context,
                &inputSM);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup4;
    }
```

```
    /* [c] Bulk output buffer (also input for digest). */
    outputSM.size  = outputSize;
    outputSM.flags = TEEC_MEM_INPUT | TEEC_MEM_OUTPUT;
    outputSM.buffer = outputBuffer;

    /* Use TEE Client API to register the underlying memory buffer. */
    result = TEEC_RegisterSharedMemory(
                &context,
                &outputSM);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup5;
    }

    /* ========================================================================
    [4] Perform cryptographic operation initialization commands
    ======================================================================== */
    /* [a] Start the encrypt operation within the TEE application. */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_VALUE_INPUT,
        TEEC_MEMREF_PARTIAL_INPUT,
        TEEC_NONE,
        TEEC_NONE);


    /* Write key ID (example uses key ID = 1) in parameter #1 */
    operation.params[0].value.a = 1;

    operation.params[1].memref.parent = &commsSM;
    operation.params[1].memref.offset = 0;
    operation.params[1].memref.size   = 16;

    /* Write IV (example uses an IV of all zeros) in to Memory buffer. */
    ivPtr = (uint8_t*)commsSM.buffer;
    memset(ivPtr, 0, 16);

    /* Start the encrypt operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_ENCRYPT_INIT,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* [b] Start the digest operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_DIGEST_INIT,
                NULL,
                NULL);

    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }
```

```
    /* ============================================================================
    [5] Perform the cryptographic update commands
    ============================================================================ */
    /* [a] Start the encrypt operation within the TEE application. */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_MEMREF_WHOLE,
        TEEC_MEMREF_PARTIAL_OUTPUT,
        TEEC_NONE,
        TEEC_NONE);

    operation.params[0].memref.parent = &inputSM;
    /* Note that the other fields of operation.params[0].memref need not be
       initialized because the parameter type is TEEC_MEMREF_WHOLE */

    /* Note: Even though we share the entire block we do so with less flags, so
     * fallback on the TEEC_MEMREF_PARTIAL method. */
    operation.params[1].memref.parent = &outputSM;
    operation.params[1].memref.offset = 0;
    operation.params[1].memref.size   = outputSize;

    /* Start the encrypt operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_ENCRYPT_UPDATE,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* [b] Start the digest operation within the TEE application. */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_MEMREF_PARTIAL_INPUT,
        TEEC_NONE,
        TEEC_NONE,
        TEEC_NONE);

    /* Note: we use the updated size in the MemRef output by the encryption. */
    operation.params[0].memref.parent = &outputSM;
    operation.params[0].memref.offset = 0;
    operation.params[0].memref.size   = operation.params[1].memref.size;

    /* Start the digest operation within the TEE application. */
    result = TEEC_InvokeCommand(
                &session,
                CMD_DIGEST_UPDATE,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* ============================================================================
    [6] Perform the cryptographic finalize commands
    ============================================================================ */
    /* [a] Finalize the encrypt operation within the TEE application. */
    result = TEEC_InvokeCommand(
```

```
                &session,
                CMD_ENCRYPT_FINAL,
                NULL,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* [b] Finalize the digest operation within the TEE application. */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_MEMREF_PARTIAL_OUTPUT,
        TEEC_NONE,
        TEEC_NONE,
        TEEC_NONE);

    operation.params[0].memref.parent = &commsSM;
    operation.params[0].memref.offset = 0;
    operation.params[0].memref.size   = 20;

    result = TEEC_InvokeCommand(
                &session,
                CMD_DIGEST_FINAL,
                &operation,
                NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup6;
    }

    /* Transfer digest in to user buffer. */
    memcpy(digestBuffer, commsSM.buffer, 20);

    /* ========================================================================
    [7] Tidyup resources
    ======================================================================== */
cleanup6:
    TEEC_ReleaseSharedMemory(&outputSM);
cleanup5:
    TEEC_ReleaseSharedMemory(&inputSM);
cleanup4:
    TEEC_ReleaseSharedMemory(&commsSM);
cleanup3:
    TEEC_CloseSession(&session);
cleanup2:
    TEEC_FinalizeContext(&context);

cleanup1:
    return result;
}
```

Table of Figures and Tables

END OF DOCUMENT