Trusted Logic®

Trusted Foundations™

Developer Reference Manual (APIs V3.x)

Date of Issue:	23 Jan 2012
Reference & Version:	CP-2010-RT-533-V2.0
Classification:	Confidential
Number of pages:	364 (including 2 header pages)

PREFACE

Version History

Version	Date	Status	Modifications
1.0	20Dec2010	Issued	First issued version
1.0.1	11Feb2011	Issued	Correction of minor typos and formatting issues
1.1	19Jan2012	Issued	Documented the usage of the Android FDM (Functional Demonstrator)
2.0	23Jan2012	Issued	Update for the version 3.1 of the Developer APIs (RSA private keys can optionally be non-sensitive)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABOUT THIS MANUAL

This manual describes how to develop software for the Trusted Foundations and provides a complete reference of the Trusted Foundations Developer APIs, version 3.x.

About the Trusted Foundations Developer APIs V3.x

The Trusted Foundations product range provides a portable Trusted Execution Environment (TEE) for running security-sensitive code. Each product utilizes hardware-based or software-based protection mechanisms to separate a “Secure World” from a “Normal World”:

- The Secure World contains the Trusted Execution Environment that runs Secure Services;
- The Normal World runs Client Applications that access the secure services.

The Trusted Foundations products include built-in services that provide off-the-shelf security functionality, such as secure data storage and a cryptographic provider. The products also allow deployment of custom services, which can, for example, implement the heart of a Digital Rights Management scheme.

The Figure 1 provides a high-level description of the product architecture and its various programming interfaces.

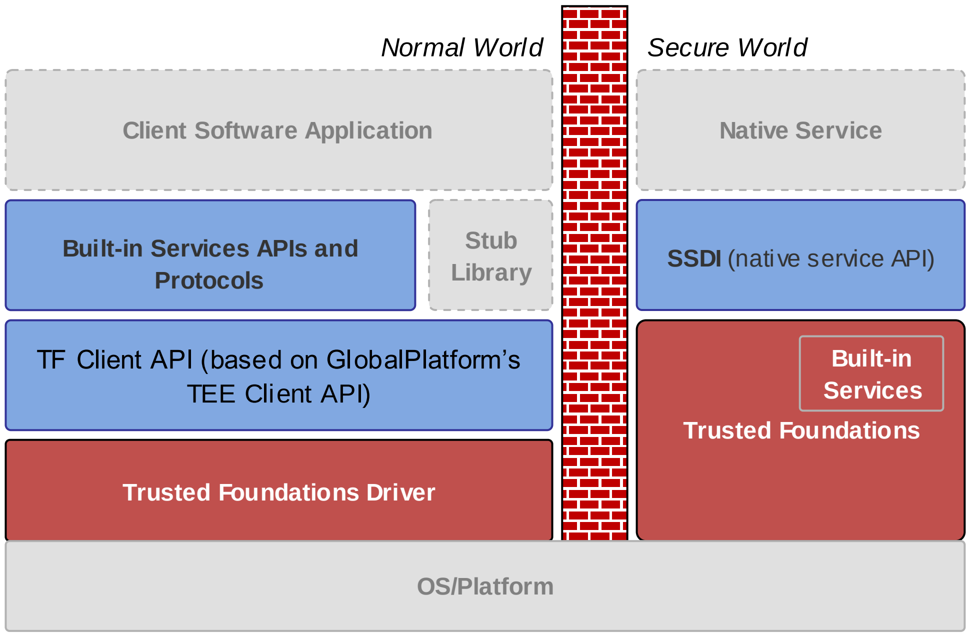

Normal World
Secure World
Client Software Application
Native Service
Built-in Services APIs and Protocols
Stub Library
SSDI (native service API)
TF Client API (based on GlobalPlatform's TEE Client API)
Built-in Services
Trusted Foundations
Trusted Foundations Driver
OS/Platform

Figure 1 High-Level Architecture of Trusted Foundations

A Trusted Foundations product typically exposes the following APIs and protocols:

- The Secure Service Development Interface (SSDI) allows the development in C of secure services;
- The TF Client API (based on GlobalPlatform's TEE Client API standard) provides a general way to access the secure services running inside the Secure World. Details of how to access a specific secure service may sometimes be hidden in a service-specific, developer-friendly API implemented by a "stub";
- Built-in services APIs and Protocols provide access to the Built-in services, which are:
    - the Cryptographic Service (PKCS#11)
    - the Secure Storage Service
    - the Monotonic Counter Service
    - the Service Manager

Collectively, all these APIs are called the Trusted Foundations Developer APIs. This manual describes how to use the Developer APIs and provides a complete reference for all the 3.x APIs. See Appendix I for more information on the version history of these APIs, in particular for a list of all the 3.x versions and their differences.

Intended Audience

This manual is written for all developers designing or implementing software to run within, or use, one of the Trusted Foundations products. It is assumed that you are an experienced embedded software developer in C.

Using this Manual

This manual is organized as follows:

- Part A, "*Using the Developer APIs V3.x*" is a general introduction to the Developer APIs. It presents the developing tools and examples included in the products
- Part B, "*Specifications of the Developer APIs V3.0*" contains the complete specification of the Developer APIs V3.x.

Part A

USING THE DEVELOPER APIS V3.X

Chapter A1 DEVELOPING FOR THE TRUSTED FOUNDATIONS

This chapter describes the development of secure services and their associated normal world software.

All Trusted Foundations products come with a Software Development Kit (SDK) located in the `tf_sdk` directory. This SDK contains:

- the C header files for compiling both the normal-world and secure-world software
- the libraries to link your code
- PC simulators (Win32 and/or Linux) to simulate the Trusted Foundations APIs
- an Android Application Package (APK) to simulate the Trusted Foundations APIs on an ARM-based Android device. This is called the Android “FDM” (for Functional DeMonstrator).
- examples

Note that the SDK shipped with your product may not contain the simulators and functional demonstrators for all operating systems.

A1.1 Building Secure World Services

A1.1.1 Overview

The source files of a service are composed of:

- Source code written in 'C' or in assembler for the target processor. These files must be compiled using the C compiler and assembler suitable for your platform.
- Service properties, which constitute the service meta-data and are composed of a set of key-value pairs of strings. These properties control various aspects of the execution of the service and can also be used by the service to configure itself or to advertise some information about itself to its clients (the full set of properties that the service can define is specified in section B2.3.6.1, "*Service Properties*"). These properties must be written in a text file, called a manifest, and compiled using a resource compiler provided with your product and described in section A1.1.2 below). This produces an object file suitable for the linker of your target platform.

Once the source files and the service properties have been compiled to object files, the linker of your target platform must be used to produce the service binary file. For each platform, the linker must be suitably configured so that it produces a file in the format recognized by the execution environment. Generally, this file format is some kind of Dynamically Linked Library (DLL).

Example

Assume a service is made of two C source files and the manifest: `service_main.c`, `service_ops.c`, and `service.manifest.txt`. Then, the typical sequence of operations to produce the service binary looks like the following:

```
1. $(CC) -o service_main.o -c service_main.c
2. $(CC) -o service_ops.o -c service_ops.c
3. tf_resc -o service.manifest.o -manifest service.manifest.txt -target
   elf
4. $(LINK) -o service_binary.srvx service_main.o service_ops.o
   service.manifest.o
```

See the product documentation for full specification of the file format of secure services.

A1.1.2 Service Resource Compiler

The `tf_resc` tool must be used to compile the manifest properties file, resources and meta-data of all native secure service written on top of the SSDI. The tool generates an object file that must be linked with the rest of the native service to produce the final binary that is suitable for integration.

In the current version of the tool, the only resource that can be compiled is the service manifest. The manifest is a list of key-value properties attached to the service which define user properties and configuration data interpreted by the system.

Command Line

The command line is the following:

```
tf_resc [options]
```

where the available options are:

- `--manifest <manifest>` : Manifest to compile
- `[-o|--output] <file.obj>` : Name of the output object file to generate
- `--target [elf | pecoff | elf386]` : File format of the output object

All options are mandatory.

Format of the manifest file

The source manifest file must be a valid UTF-8 text file that contains a set of properties represented as “name: value” pairs organized in sections. It is acceptable for the string of Unicode characters encoded by the UTF-8 file to start with the character U+FEFF (BOM character) which can be inserted by some text editors to denote an UTF-8 file. Once the optional leading U+FEFF character has been removed, the remaining Unicode characters must comply with the following BNF syntax:

```
property-file:   +(*blank-line property) *blank-line
blank-line:      comment | *space newline
comment:         HASH *otherchar newline
newline:         CR LF | LF | CR
space:           SPACE | TAB
property:        name *space COLON *space value *space newline
name:            alphanum *headerchar
value:           *otherchar
alphanum:        {A-Z} | {a-z} | {0-9}
headerchar:      alphanum | MINUS | UNDERSCORE | DOT
otherchar:       any UTF-8 character except NUL, CR and LF
CR:              <Unicode CR (U+000D)>
LF:              <Unicode LF (U+000A)>
SPACE:           <Unicode SPACE (U+0020)>
TAB:             <Unicode TAB (U+0009)>
NUL:             <Unicode NUL (U+0000)>
MINUS:           <Unicode MINUS (U+002D)>
UNDERSCORE:      <Unicode UNDERSCORE (U+005F)>
DOT:             <Unicode DOT (U+002E)>
HASH:            <Unicode HASH (U+0023)>
COLON:           <Unicode COLON (U+003A)>
```

The manifest file is a list of properties separated by blank lines or comments, where:

- each property has a name that must start with a letter or a digit and can continue with numbers, digits, dashes ‘–‘, underscores ‘_’ and dots ‘.’;
- each property has a value that can contain any characters, except newlines and NUL. Note that the value is stripped of any leading or trailing spaces and tabs by the implementation when read

The final line of the configuration file is not required to end with a *newline* sequence.

The manifest must not define two properties with the same name.

The manifest file must contain a property named “config.s.serviceID”. Its value must be the valid string representation of a Universally Unique Identifier as defined in **[RFC4122]**, which is interpreted case-insensitive. This is the identifier of the service.

Here is an example of a correct manifest file describing a multi-instance and multi-command service and containing two service-specified properties (“name” and “version”):

```
config.s.serviceID: 800730f9-cf5d-43d5-898f-6e5940bcaebb <CR> <LF>
config.s.multi_instance: true <CR> <LF>
config.s.multi_command: true <CR> <LF>
name: DRM Service <CR> <LF>
version: 1.0 <CR> <LF>
```

For more details on the specification of the service properties, refer to section B2.3.6.1, “*Service Properties*”.

Output formats supported

The `--target` option allows the caller to decide what binary format the output object will have. Select an appropriate target for the platform you are developing for:

- `elf`: Use this format for ARM-targeted Trusted Foundations (ARM GCC or RVCT) and Android FDM.
- `pecoff`: Use this format for Trusted Foundations Windows PC Simulator

- `elf386`: Use this format for Trusted Foundations Linux PC Simulator (ix86 Linux)

A1.1.3 INTEGRATING A SERVICE

Depending on the execution environment used to run the services, the service binary can either be used as is, or must be further processed to be usable by the environment. For example, on some platform, service binaries must be postlinked with the master binary of the execution environment.

See the product integration guide for full details on secure services integration to the core binary.

The services can also be run in PC simulators. See section A1.3, "*Using the PC Simulators*" for more details.

A1.2 BUILDING NORMAL WORLD SOFTWARE

A1.2.1 HEADER FILES AND LIBRARIES

The following header files are provided with your SDK:

- `cryptoki.h`: definitions for the External Cryptographic API
- `sst.h`: definitions for the External Secure Storage API
- `mtc.h`: definitions for the External Monotonic Counter API
- `ssdi.h`: definitions for SSDI
- `tee_client_api.h`: definitions for the TF Client API

The following static libraries or shared objects are provided for linking. There is a copy for each supported Normal World Operating System:

- `tee_client_api_`*`<impl>`*`.lib` / `libtee_client_api_`*`<impl>`*`.so`: link library for the TF Client API. The precise library to select depends on the targeted Trusted Foundations instance (simulator or real integration)
- `tf_crypto_sst.lib` / `libtf_crypto_sst.so` : link library for the External Cryptographic API, the External Secure Storage API, and the External Monotonic Counter API

A1.2.2 CLIENT SIGNING TOOL

If you want to use the Client Authentication feature (see section B3.2.6), you must generate a signature file for you client application.

A reference implementation of a client signing tool is provided in the form of a simple script named `tf_client_sign.bat` on Windows and `tf_client_sign` on Linux. This script relies on OpenSSL and Python. For your convenience, a pre-built version of these tools is included in the package.

Note that the `tf_client_sign` script is not intended to be used in a production environment but is meant as a reference implementation to generate the signature format specified in section B3.3.6.

Refer to the examples that use Client Authentication to see how to use this reference script.

All the products are preconfigured with a 1024-bit RSA default development public key. The corresponding RSA private key is the following:

```
modulus:
0x00a23932d579a3169f4018ccb3a3dcded2564ca5b4313187d3580de2e6c3c8cd591
7381cce29ea389f23b051d0f33910af8cbbd6dd97fc50e5876aff33738a4edebcb070
6d30019f8f924d4a40175838f54afe471ce8494976c8117bc885cd55fc125a61cae7f
0a078943cfd71ffd5096da5a5cc0029aba0882092bac3dadf669f
publicExponent:
0x10001 (65537)
privateExponent:
0x1bd5aea83c67c460ecd194861a4eb298ea9fda2ad1ea0ac133b1a6462c3fcf24433
ca88ca35a81fea45403a5d3a5edfc012cd5d04ddcc850a9ecacff3a0af83f2f9937ed
8e5ff5bf2c19532b14acf43e7f9aba5aec4a343ce098c0b35cfc995caa19c414054d5
b845925cf1c31f406dc7bd25d308f796a4d45203520f0ba2c91
prime1:
0x00d81de5637156caa48875d957669f578cf67bafd45131aa2afd79a2d77f5d83186
69c1a44675da2e8774c3eca7c1a6841133f9336fd2ed0f715654e6dc5680397
prime2:
0x00c02931c197ee5fec1ca84f5da9f5c1fb091ae4ec259a3d5b351f460e03519b15a
d30c38ec1ed0293fda8239eb9bf35af8ccd6b1985807b2e6f902cc569497639
exponent1:
0x00c6d8db12e059c722141cbad95c27085b3eff170cf79806c67f6ac796182e664cb
13cb83e700d0bb370f0ca656c42afe2105fd28829f44578d24ae01ac823a809
```

```
exponent2:
0x371517e0288efa0c1282e48d32c4f19fe6124180b79ea8ebd6246ace658124f567b
9ccb4c53e0d3922c2e9c2fbf7a589f6d0835cc379fad56ac1673407643971
coefficient:
0x4c176188482c1959761420b70642b936419466ad9aeaf4c53255a3fce939e53a702
0f9f2830b9522f8cdf3ca94046dfb0d06e505d23c45d1346a6122807ae30c0
```

A1.3 USING THE PC SIMULATORS

A1.3.1 OVERVIEW

Trusted Foundations products contain PC simulators running on Windows and/or Linux. The role of the Trusted Foundations PC Simulators is to provide the same Developer APIs as the products, but in a simulation environment running on a PC under Windows or Linux. The PC simulators do not interface any specific security hardware.

The Trusted Foundations PC simulators support secure services developed with SSDI using C and compiled to native X86 machine code.

A1.3.2 USER GUIDE

There are two versions of the Trusted Foundations PC simulators:

- The Win32 simulator is a simulation version of the Trusted Foundations running on a Win32 PC (ix86 architecture). This simulator is located in the directory `simulator_ix86_win32` in the SDK.
- The Linux simulator is a simulation version of the Trusted Foundations running on a Linux PC (ix86 architecture only). This simulator is located in the directory `simulator_ix86_linux` in the SDK.

These tools allow the functional development of applications, stub libraries, and secure services using the APIs of the Trusted Foundations.

A1.3.2.1 SUPPORTED FUNCTIONALITY

Refer to the *Product Reference Manual* shipped with your product for full details of supported functionality, such as precise cryptographic algorithm availability.

A1.3.2.2 INSTALLATION

Unzip the SDK in any directory on the PC. It is recommended not to use spaces in the name of this directory.

For the TF Linux simulator, you may have to set the execution permission of the TF simulator binary files. The TF simulator binaries are located in the folder `simulator_ix86_linux` of the SDK.

A1.3.2.3 COMMAND LINE

To start the simulator, with selected services *`serviceX`* (if needed), use the following command line:

```
tf_simu [options] service1 . . . serviceN
```

The following options are defined for both Win32 and Linux simulators:

- `[-h|--help]` : Display the online help
- `[-storageDir <path>]` : Directory in which persistent data are stored (optional)
- `[--perso]` : Format the file system and keystores on startup (optional)
- `[-c <path>]` : Path to configuration file (optional)
- `[-deviceName <name>]` : Use the given name instead of the default device name.

Supported Secure Service Formats

The files for *`serviceX`*, identified on the command line by their filename path, must be Secure Services compiled for the ix86 Linux (for Linux simulator) or Win32 (for Win32 simulator).

Configuration File

This section specifies the configuration file that can be passed to the PC simulators (option `-c`).

Syntax

The Configuration File must be a valid UTF-8 text file that contains a set of properties represented as “name: value” pairs organized in sections. It is acceptable for the string of Unicode characters encoded by the UTF-8 file to start with the character U+FEFF (BOM character) which can be inserted by some text editors to denote an UTF-8 file. Once the optional leading U+FEFF character has been removed, the remaining Unicode characters must comply with the following BNF:

```
configuration-file: +section *blank-line
section:            *blank-line OPEN_BRACKET name CLOSE_BRACKET newline properties
properties:         *(*blank-line property *blank-line)
blank-line:         comment | *space newline
comment:            HASH *otherchar newline
newline:            CR LF | LF | CR
space:              SPACE| TAB
property:           name COLON *space value *space
name:               alphanum *namechar
value:              *otherchar
alphanum:           {A-Z} | {a-z} | {0-9}
namechar:           alphanum | MINUS | UNDERSCORE | DOT
otherchar:          any UTF-8 character except NUL, CR and LF
CR:                 <Unicode CR (U+000D)>
LF:                 <Unicode LF (U+000A)>
SPACE:              <Unicode SPACE (U+0020)>
TAB:                <Unicode TAB (U+0009)>
NUL:                <Unicode NUL (U+0000)>
MINUS:              <Unicode MINUS (U+002D)>
UNDERSCORE:         <Unicode UNDERSCORE (U+005F)>
DOT:                <Unicode DOT (U+002E)>
HASH:               <Unicode HASH (U+0023)>
COLON:              <Unicode COLON (U+003A)>
OPEN_BRACKET        <Unicode LEFT SQUARE BRACKET (U+005B)>
CLOSE_BRACKET       <Unicode RIGHT SQUARE BRACKET (U+005D)>
```

It is forbidden to have duplicate property names within a section; however the same property name can be used in different sections. The names of sections are not case sensitive. The names of properties are case sensitive. The value of each property is stripped of any leading and trailing whitespaces.

Types

Each property value is a Unicode string of characters. However, some properties are interpreted as having particular types, like integers or Booleans:

- Integers can be expressed in decimal, hexadecimal, or binary form:

```
integer:     decimal-integer | hexadecimal-integer | binary-integer
decimal-integer:    +[0-9]
hexadecimal-integer: 0[xX]+[0-9a-fA-F]
binary-integer:      0[bB]+{0,1}
```

  Integers are always unsigned and must be representable in 32 bits, so they can range from 0 to 2^{31}-1. If a property marked with the Integer type does not satisfy this constraint, the simulator considers the configuration file as invalid.

- Boolean properties must have the value “true” or “false”, case-insensitive.

Supported Properties

The simulator configuration file contains only optional properties that can be set in the section named “`Global`”. They are defined in the following table:

Name	Type	Description
`workspace.1.size`	Integer (optional)	Size in bytes of the secure workspace memory segment. Must be a multiple of 4kB.
`clientAuthentication.rootKey`	String (optional)	Indicates the path to the file containing the RSA public key for client authentication. This path is relative to the configuration file. This parameter is

		optional. If not specified, the default developer key will be used to authenticate clients. If specified, the key must be an RSA public key up to 2048-bits in length. The key must be represented as an RSAPublicKey object as defined in **[PKCS#1]** and encoded using DER. The DER-encoded key must be put in a file that is referenced in this clientAuthentication.rootKey property.

Configuration File Sample

This is an example of a configuration file:

```
#
# Trusted Foundations Simulator Configuration File (Example).
#

[Global]

# Global Memory (size in bytes)
workspace.1.size: 0x1000000
```

Device Name Usage

The device name specified by the option -devicename determines the communication channel used to communicate between Normal-World and Secure-World. This implementation supports two communication channels: TCP sockets and UNIX local sockets (for the Linux simulator only).

The device name must be of the form:

- `TCP://<IPAddress>:<Port>`, where `<IPAddress>` is the IP address (for example 127.0.0.1) and `<Port>` is the port number (for example 3101) the simulator must listen to.
- (Linux simulator only) `UNIX://<File>`, where `<File>` is the file local socket (for `example /tmp/smapi_socket`) the simulator must listen to. Note that `<File>` should begin with the character '`/`'.

The implementation selects the device name to use as follows:

- If the option `-devicename` is specified, the value specified by this option is used. If this value is not correctly formed, the simulator fails to launch.
- If the option `-devicename` is not specified and the environment variable "`TFSW_URL`" is defined, then the value of the environment variable "`TFSW_URL`" is used. If this value is not correctly formed, the simulator fails to launch.
- If the option `-devicename` is not specified and the environment variable "`TFSW_URL`" is not defined, then a default value is used. The default value used is: `TCP://127.0.0.1:3001`

On the Normal-World side, the device name used by the TFAPI must match the device name used by the simulator. The device name used by the Normal-World is determined as follows:

- The value of the device name can be coded in the client application or library that calls the TF Client API. In such a case, this device name is used and cannot be changed (unless changing the client application or library).
- Otherwise, if no application or library specifies a value for the device name, and if the environment variable "`TFSW_URL`" is defined, then the value of the environment variable "`TFSW_URL`" is used.
- Otherwise, if no application or library specifies a value for the device name, and if the environment variable "`TFSW_URL`" is not defined, then a default value is used. The default value used is: `TCP://127.0.0.1:3001`.

A1.4 Using the Android FDM

A1.4.1 Overview

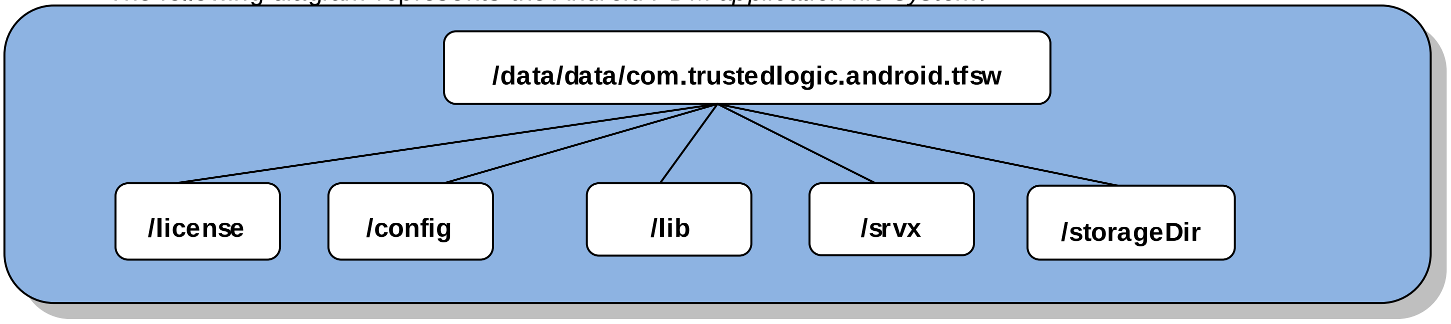
The Trusted Foundations Android FDM (Functional DeMonstrator) is an installable Android Application providing the same Developer APIs as the products, but in a simulation environment running on an Android Emulator or on a real Android device. Android version 1.5 or beyond is required to use the Android FDM.

The Android FDM is an installable APK and does not interface any specific security hardware.

The FDM exposes the Trusted Foundations built-in services (cryptographic service, secure storage, monotonic counter, the service manager) and can also work with user-defined services developed with SSDI using C and compiled to native ARM code using GCC. These services are implemented as native libraries which are loaded by the Android FDM when it is started. The Functional Demonstrator for Android is implemented through an Android Service (invisible and running in the background), but it can be managed (started/stopped) by its associated Activity (interface).

The communication between the TF Client API and the Android FDM is based on TCP sockets.

The following diagram represents the *Android FDM application file system*.



The file system of the application (*com.trustedlogic.android.tfsw*) contains five directories:

- **license** contains the license file “`TFSW.license`”;
- **config** contains the the configuration file “`TFSWparameter.cfg`”;
- **lib** contains the Trusted Foundations implementation as a native shared library;
- **srvx** contains user-defined services;
- **storageDir** is the directory where persistent data are stored.

A1.4.1.1 Installation & Configuration

The Android FDM APK can be installed on the device (real or emulated) via Android Debug Bridge. ADB is a tool included in the Android SDK which can be downloaded from the following location: http://developer.android.com/sdk/index.html.

A DOS batch file which automatically performs the installation of the APK through ADB can be found in `tf_sdk/fdm_android/install_trustedfoundations.bat`.

Once the Android FDM is installed, it will be launched automatically at boot-time. The user can manage it through the Trusted Foundations Software manager activity (Trusted Foundations Software icon), which displays service status and permits to start or stop it.

A1.4.1.2 Licence File

The software requires the licence file named “`TFSW.license`”. This file is provided by Trusted Logic and used to control the distribution of the software. It can be stored in several locations and is retrieved in the following order at startup.

1. If the `TFSW.license` file exists in the sdcard/tfsw directory, it is automatically installed in the `data/data/com.trustedlogic.android.tfsw/license` directory (any existing file is overwritten) and is used.

2. Else if the `TFSW.license` file exists at the root of the sdcard directory, it is installed in the `data/data/com.trustedlogic.android.tfsw/license` directory (any existing file is overwritten) and is used.
3. Else if the `TFSW.license` file exists in the `data/data/com.trustedlogic.android.tfsw/config` directory, it is used.
4. Else there is no valid license, and thus the software will not work.

A1.4.1.3 PARAMETERS FILE

The software can be configured via a configuration file named "`TFSWparameter.cfg`". This file can be located in several locations and is retrieved in the following order:

1. If the `TFSWparameter.cfg` file exists in the `sdcard/tfsw` directory, it is automatically installed in the `data/data/com.trustedlogic.android.tfsw/config` directory (any existing file is overwritten) and is used.
2. Else if the `TFSWparameter.cfg` file exists at the root of the sdcard directory, it is installed in the `data/data/com.trustedlogic.android.tfsw/config` directory (any existing file is overwritten) and is used.
3. Else if the `TFSWparameter.cfg` file exists in the `data/data/com.trustedlogic.android.tfsw/config` directory, it is used.
4. Else the TFSW is started with the default parameters and just the built-in secure services (Cryptographic Service, Secure Storage Service, Monotonic Counter Service, and Service Manager).

The file `TFSWparameter.cfg` contains the parameters for the Android FDM. This file is loaded using the load() method of the class java.util.Properties. The file format must comply with this method. That is (extracted from the Android documentation):

"The encoding is ISO8859-1. The Properties file is interpreted according to the following rules:

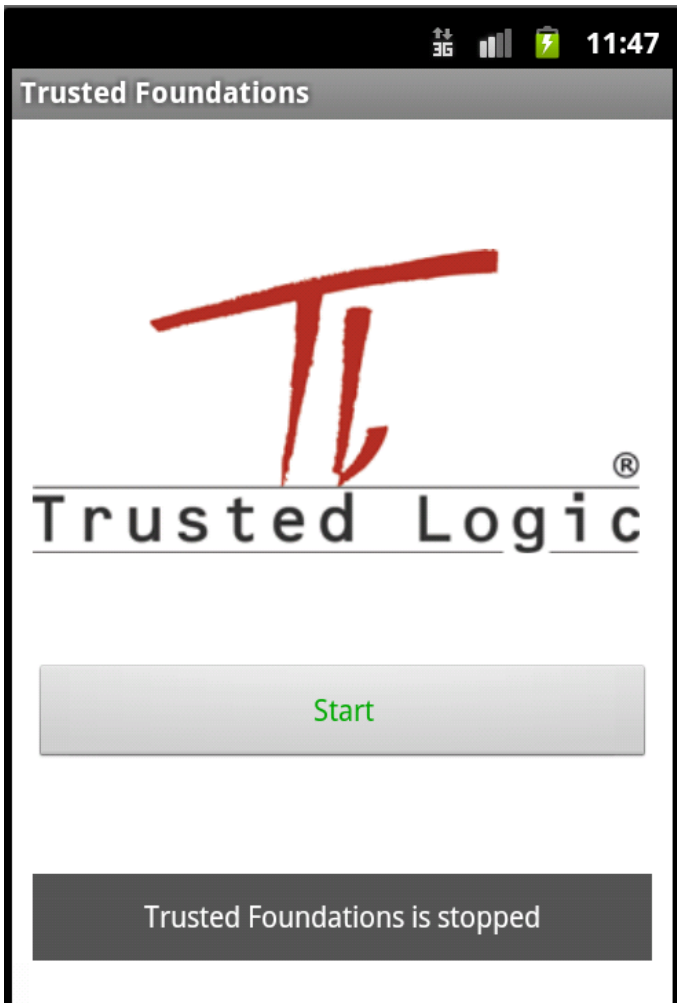
- *Empty lines are ignored.*
- *Lines starting with either a "#" or a "!" are comment lines and are ignored.*
- *A backslash at the end of the line escapes the following newline character ("\r", "\n", "\r\n"). If there's a whitespace after the backslash it will just escape that whitespace instead of concatenating the lines. This does not apply to comment lines.*
- *A property line consists of the key, the space between the key and the value, and the value. The key goes up to the first whitespace, "=" or ":" that is not escaped. The space between the key and the value contains either one whitespace, one "=" or one ":" and any number of additional whitespaces before and after that character. The value starts with the first character after the space between the key and the value.*
- *Following escape sequences are recognized: "\ ", "\\", "\r", "\n", "\!", "\#", "\t", "\b", "\f", and "\uXXXX" (unicode character)."*

The following properties can be configured:

- **`deviceName`**: `TCP://`*`<IPAdress>`*`:`*`<Port>`* where *`<IPAddress>`* is the IP address (for example 127.0.0.1) and *`<Port>`* is the port number the Android FDM must listen to.
  - If this property is missing, the default port number (3001) is used.
- ***`<myService>`***`.srvx`: *`<serviceUrl>`* defines the URL of a secure service file that must be loaded with the Android FDM.
  - *`<myService>`*: can be any String.
  - *`<serviceUrl>`*: must be of the following form:
    - `sdcard/`*`<pathToSRVX>`*: the complete path to an srvx file on the SD card. For example: `sdcard/tfsw/myservice.srvx`

A1.4.1.4 LAUNCHING TRUSTED FOUNDATIONS SERVICE

Once the Android FDM is installed, it will be launched automatically at boot-time. The user can manage it through the Trusted Foundations manager activity. The activity is launched from the "Applications" menu and displays a user interface as below which which displays the Trusted Foundations status (running/stopped) and allow us to toggle it.



Trusted Foundations is stopped

A1.4.1.5 INSTALLING YOUR NORMAL WORLD CLIENT

Your application package (APK) is to be installed as per any other on Android i.e. using Android Debug Bridge, or the Eclipse Android plug-in of the Android SDK, or an application of type "apkInstaller" (available from Android Market).

Your client can be coded in Java and calling the native stubs via JNI, or be fully coded in C (if running on Android 2.3 or later).If using a Java wrapper, the native stub shared library must be included in the libs/armeabi folder of your project so that it is included in your APK.

All examples in the TF SDK use the JNI method with Java Wrapper (meaning they can be installed on Android devices 1.5 or later).

A1.4.1.6 INSTALLING YOUR SECURE SERVICE

Since "sdcard" is the only common shared space on an Android device, it is used for provisioning the Android FDM. On startup, the Android FDM will copy necessary files from "sdcard" to its internal application structure as described above.

1. Compile your service's manifest file with the tf_resc tool in the TF SDK, for example:

```
tf_resc.exe --output service.manifest.o --manifest service.manifest.txt
--target elf
```

2. Compile your service using toolchain included in the Android NDK and link with the manifest object file to make a shared library.
3. Add an entry to TFSWparameter.cfg, for example:

```
TestService.srvx:/sdcard/TestService.srvx
```

4. Copy `TFSWparameter.cfg` to `/sdcard` (or `/sdcard/TFSW/`), for example., using adb:

   ```
   adb push TFSWparameter.cfg /sdcard/TFSW/TFSWparameter.cfg
   ```

5. Copy your service to `/sdcard` or (or `/sdcard/TFSW`) e.g. using adb:

   ```
   adb push MyService.srvx /sdcard/TFSW/MyService.srvx
   ```

6. (Re)start the Android FDM on the device,as described in section A1.4.1.4, so that the new services is taken in to account. The SRVX file will the be found via the `TFSWparmater.cfg` file and copied in the `srvx` directory in the Android FDM application directory
7. Verify your service is present (via TFCTRL example, by launching your client etc.)

A1.4.2 DEVELOPMENT TOOLS

Secure Services (`.srvx`) are written C, compiled with the Android NDK (as "Shared Library") and linked with `tf_sdk/lib/fdm_android/libssdi.so`.

Normal World clients are either written in Java calling native stubs via JNI or fully coded in C (this requires Android version 2.3 or later) and linked with `tf_sdk/lib/fdm_android/libtee_client_api_socket.a`. If you want to use the built-in secure services (Cryptographic, Secure Storage, or Monotonic Counter), then you must link the code with `tf_sdk/lib/fdm_android/libtf_crypto_sst.so`.

A1.4.2.1 ANDROID SDK & NDK

The Android SDK (Standard Development Kit) and the Android NDK (Native Development Kit) are needed to build new applications / services. Note the native libraries created by the Android NDK can only be used on devices running the Android 1.5 platform version or later. This is due to toolchain and ABI related changes that make the native libraries incompatible with 1.0 and 1.1 system images.

A1.4.2.2 ECLIPSE

Eclipse is very useful to compile APK files, manage an emulator and install applications on real or emulated device). The Android SDK includes an Eclipse plugin to ease development and debugging.

In the "DDMS" perspective, SSDI traces (functions `SLogTrace`, `SLogError`, and `SLogWarning`) are routed via the Android logging system ("LogCat") and can therefore be viewed, filtered etc. from within Eclipse.

A1.4.2.3 SERVICE INSTALLER

A DOS batch script is available in the product package at `tf_sdk/fdm_android/install_services.bat` which automates the service installation procedure described in section A1.4.1.6 for all .srvx files found in the current directory.

A1.4.2.4 TFCTRL APK

The product package contains an installable APK which implements the TFCTRL example (see section A2.2). This APK can be installed on the device to verify that your secure services (post-linked or otherwise) are visible to Android FDM and therefore, to Normal World clients.

The APK is located at `tf_sdk/examples/example_tfctrl/example_tfcrl.apk`. There is also a DOS batch file "`install_tfctrl_apk.bat`" for installation via ADB along with the necessary DOS batch and make files to rebuild it using Android NDK and Android SDK.

Once installed, a TFCTRL icon will be added to the devices "Applications" menu so that it can be launched.

Chapter A2 EXAMPLES

This chapter provides a detailed documentation for the examples included in the products.

- The section A2.1 provides general instructions to build and execute the examples.
- The subsequent sections each describe one example;
  - section A2.2, "*TFCTRL Example*" shows a simple usage of the TFAPI;
  - section A2.3, "*External Crypto* Example" shows how to use the External Cryptographic API;
  - section A2.4, "*External Secure Storage Example*" shows how to use the External Secure Storage API;
  - section A2.5, "*External Monotonic Counter* Example" shows how to use the External Monotonic Counter API;
  - section A2.6, "*DRM Example*" shows an example of Secure Service that uses SSDI;
  - section A2.7, "*CryptoCatalogue Example*" shows a Secure Service that makes use of many typical cryptographic operations using the Internal Cryptographic API;
  - section A2.8, "*CertStore Example*" shows a Secure Service that makes use of the Secure Storage and Cryptographic capabilities of SSDI;
  - section A2.9, "*Secure Date Example*" shows a Secure Service that makes use of the Secure Date capabilities of SSDI.

A2.1 GENERAL INSTRUCTIONS TO BUILD AND EXECUTE THE EXAMPLES

This section provides general information on how to build and execute the examples included in the products SDKs.

A2.1.1 HOW TO BUILD AN EXAMPLE

Each example, denoted `exampleXXX` in the following sections, is located in the directory `examples/exampleXXX` in the SDK.

- Normal world components:
  - a client application, denoted `clientXXX` in the following sections
  - sometimes, a secure service stub, denoted `stubXXX` in following sections (for instance for the DRM example). Not all examples include a stub. Most directly access the TF Client API from the client;
- Sometimes a secure world component:
  - a secure service, denoted `serviceXXX` in following sections (for instance for the DRM or CryptoCatalogue example). Not all examples include a service. Some only use the Normal World 'External' APIs.
- Build scripts, projects, or Makefiles for the targets supported in your product.

Building the examples depends on the targeted environment:

- for a PC simulator and for the Android FDM, both the Normal World and Secure World components can be built using a single script;
- For an embedded target (not a PC simulator), the Normal World components and the Secure World components must be built separately using different scripts.

A2.1.1.1 WIN32 PC SIMULATOR

The build projects delivered are Microsoft Visual Studio 2008 projects. They can be used with the free 'Express' edition of the Visual C++ that you can download at http://www.microsoft.com/express/:

1. Open the `exampleXXX.sln` project file in the build directory with Microsoft Visual Studio. It contains projects for the different components:
   - the client application `clientXXX`;
   - the secure service stub `stubXXX` (when applicable, for instance for the DRM example);
   - the secure service itself `serviceXXX` (when applicable, for instance for the DRM example);
2. Execute the "Build / Build Solution" option from the build menu;
3. This will generate:
   - the client application `clientXXX.exe`;
   - the signature file for the client `clientXXX.exe.ssig` when the example makes use of client authentication;
   - the stub libraries, `stubXXX.lib` and `stubXXX.dll` when applicable
   - the service binary `serviceXXX.srvx` when applicable. Note that `.srvx` is the extension for a compiled secure service. It is actually a Win32 DLL.

A2.1.1.2 LINUX PC SIMULATOR

A Linux makefile is provided to rebuild each example `exampleXXX`.

You must have GCC installed on your development PC and accessible

1. Open a shell prompt and go into the directory that contains the makefile of the example;
2. Execute the "`make`" command;
3. This will generate in the release directory (default configuration):
   - the client application `clientXXX`;

- o the signature file for the client `clientXXX.ssig` when the example makes use of client authentication;
- o the stub library, `stubXXX.so`, when applicable
- o the service binary `serviceXXX.srvx` when applicable. Note that `.srvx` is the extension for a compiled secure service. It is actually a Linux Shared Object.

A2.1.1.3 ANDROID FUNCTIONAL DEMONSTRATOR (FDM)

For the Android functional demonstrator (FDM), build scripts are included which use the Android NDK and the Android SDK.

1. Open a DOS command shell
2. Set the `%ANDROID_NDK_ROOT%`, `%CYGWIN_ROOT%`, and `%PRODUCT_ROOT%` environment variables as per your configuration
3. Go to the `build` directory of the example
4. Launch the one of `build_`*`client`*`.bat / build_service.bat` file

The corresponding executable, (build_client) or .srvx (build_service) will be generated and placed in the current directory.

For an example of srvx generation, see the CryptoCatalogue example in section A2.7.

The APK can then be generated either by using the `build_apk.bat` file, or through Eclipse. If launched from the .bat file, the generated APK will be placed in the current directory.

It is recommended the project be opened and built with Eclipse to take advantage of Android SDK plugin which eases APK generation and debug (via SSDI traces in DDMS perspective).

A2.1.1.4 BUILD OF THE SECURE SERVICE

For an embedded target (not a simulator or a functional demonstrator), when an example includes a Secure Service, you must have a PC under Windows with RVCT 4.0 installed to build it. Assuming that ARMCC and ARMLINK are available in the path:

1. Open a command shell
2. Navigate into the `build` directory of the example
3. execute the command:
   ```
   > build_serviceXXX.bat
   ```
4. This will generate `serviceXXX.srvx` (`.srvx` is the extension for secure services)

A2.1.1.5 BUILD OF NORMAL WORLD COMPONENTS FOR LINUX

If your Normal World operating system is Linux, you must have a PC running on Linux with ARM Linux installed.

A makefile is provided with each example which can be run from the command shell. This makefile has been tested with GNU Make, and assumes that `arm-none-linux-gnueabi-gcc` and `arm-none-linux-gnueabi-ld` are available from the command line.

1. Open a command shell
2. go to the `build` directory of the example
3. execute the following command:
   ```
   > make
   ```
4. this will generate the client executable (`clientXXX`). It will also generate the stub library (`stubXXX.so`), when applicable

A2.1.1.6 BUILD OF NORMAL WORLD COMPONENTS FOR ANDROID

These instructions apply if your Normal World operating system is Android, but not if you use the Android FDM.

You must have a PC running on Linux with ARM Linux installed. It is assumed that you have already built an Android File System from your Android repository.

1. Open a command shell;
2. Set the `ANDROID_ROOT` environment variable to your root Android File System Repository directory. It corresponds to the directory where you entered the "`make`" command to build your Android File System;
3. Set the `ANDROID_PRODUCT` environment variable to your output product directory. It may be similar to "`$ANDROID_ROOT/out/target/product/ldp1`";
4. Navigate into the `build` directory of the example
5. Execute the script:
   ```
   > build.sh
   ```
6. This will generate:
   - the client application `clientXXX`;
   - the signature file for the client `clientXXX.ssig` when the example makes use of client authentication;
   - the stub library, `stubXXX.so` when applicable
7. The output binaries are placed in `$ANDROID_PRODUCT/system/bin` and `$ANDROID_PRODUCT/system/lib`

A2.1.1.7 BUILD OF NORMAL WORLD COMPONENTS FOR SYMBIAN

To build the client application and secure service stub, you must have the Symbian SDK and Carbide C++ installed on your development PC:

1. Open a command shell window.
2. Go into the directory that contains the project for the normal world components.
3. Build the project by executing the following commands from the command shell:

```
> bldmake bldfiles
> abld build winscw
```

This will generate `clientXXX.exe`, `clientXXX.exe.ssig` and `stubXXX.dll` (when applicable)

A2.1.2 HOW TO EXECUTE AN EXAMPLE

Start the Secure World

If you use a Trusted Foundations Simulator, refer to the Trusted Foundations Simulator description (see section A1.3, "*Using the PC Simulators*") with the service `serviceXXX.srvx` (when applictable).

If you use the Android Functional Demonstator, it is started automatically when the device boots. Otherwise it can be started and stopped via the associated activity as described in section A1.4.1.4, "*Launching Trusted Foundations service*".

If you use another product, refer to the product integration guide to start the Secure World, with the service `serviceXXX.srvx` (when applicable).

Start the Example Client Application

If the program is started from the command line interface on the platform, ensure that the command shell option for your OS is enabled on your development platform (this is not always the case).

To start the example client application:

- Open a command shell window, and navigate into the directory that contains the executable file of the client application, denoted `clientXXX` or `clientXXX.exe`.depending on the target
- Run the client application, with one of the command line (depending on the target):
  ```
  > clientXXX [options] <parameters>
  ```
  or
  ```
  > clientXXX.exe [options] <parameters>
  ```
  The options and parameters are described in each example.

For the Android FDM, the TF SDK includes APK versions of some of the examples. When an example is available as an APK, it can be launched from the “Applications” menu using the associated Activity icon. If the examples APKs are launched without the Secure World running, they will not succeed.

All examples are also provided in native executable format and can be launched from the adb shell. Examples that use the external Cryptographic, Secure Storage, or Monotonic Counter APIs require access to the `libtf_crypto_sst.so` shared library. Also, all examples require the permission to create INET sockets so as they can successfully connect to Trusted Foundations.

A2.2 TFCTRL EXAMPLE

This section concerns the TFCTRL Application Example. It shows how to use the Service Manager Protocol, specified in Chapter B7, “*Service Manager Protocol*”, through the TF Client API, specified in Chapter B3, “*Trusted Foundations Client API*”.

A2.2.1 OVERVIEW

This is an example application that accesses the Service Manager through the TF Client API.

This example application executes a sequence of operation on the Service Manager using the TF Client API to communicate with the Service Manager.

It lists the TF implementation properties, all the secure services installed within the TF, and the list of their properties.

The following figure depicts the main components of the architecture interacting in this example:

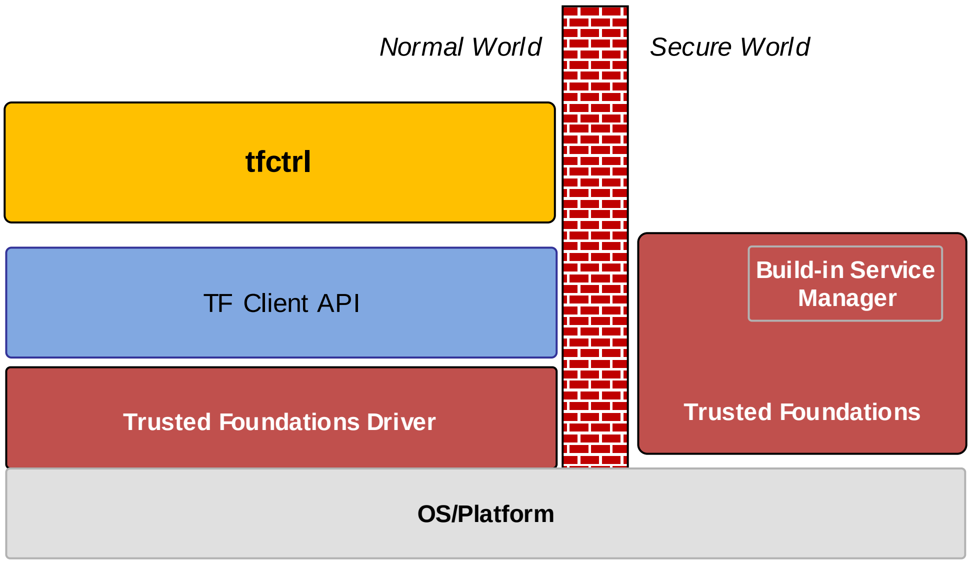


Figure A2-1: TFCTRL Example Architecture.

A2.2.2 USER GUIDE

How to build the example

This example is located in the directory `examples/example_tfctrl` in the SDK.

In this example, only a Normal World component (the client application `tfctrl`) must be built.

Refer to section A2.1.1 to know how to build the example in a specific target.

Execution of a client request

First, start the Secure World.

Then start the example application:

- Open a command shell window, and navigate into the directory that contains the executable file `tfctrl` (or `tfctrl.exe`, depending on the target).
- Execute “`tfctrl list`” to run the TFAPI example (`tfctrl` alone allows to see the available list of commands ; only `list` is supported).
- You should observe all the Trusted Foundations implementation properties, all the secure services installed within the Trusted Foundations and the list of their properties such as:

```
Trusted Logic TFCTRL
```

```
[Implementation Properties]
 apiDescription  : <TF client API description>
 commsDescription: <Communication subsystem description>
 TEEDescription  : <TEE description>

[56304B83-5C4E-4428-B99E-605C96AE58D6]
smx.name: SERVICE_SYSTEM
```

For instance, for the TF Win32 simulator, it could be:

```
Trusted Logic TFCTRL

[Implementation Properties]
 apiDescription  : TFOWX08.01.28170
 commsDescription: TFOWX08.01.28170
 TEEDescription  : TFOWX08.01.28170

[56304B83-5C4E-4428-B99E-605C96AE58D6]
smx.name: SERVICE_SYSTEM
```

A2.3 External Crypto Example

This section describes the External Crypto Example. It shows a simple usage of the External Cryptographic API, specified in Chapter B4, “*Trusted Foundations Cryptographic API*”.

A2.3.1 Overview

This example demonstrates an application accessing the External Cryptographic API. The application performs a sequence of operations that are typical of those performed by common DRM solutions when using directly the External Cryptographic API.

Another version, with the equivalent functionality but using a secure service is demonstrated in the DRM example (see section A2.6).

The application loads some encrypted content and an associated license file containing the decryption key, which is itself encrypted using an asymmetric key algorithm. It first decrypts the license key using private half of an RSA key pair, and then uses the license key to decrypt the content file. The resulting clear-text is displayed on screen.

This process is illustrated in the following figure showing also the steps that have been performed in order to produce the encrypted content and licenses:

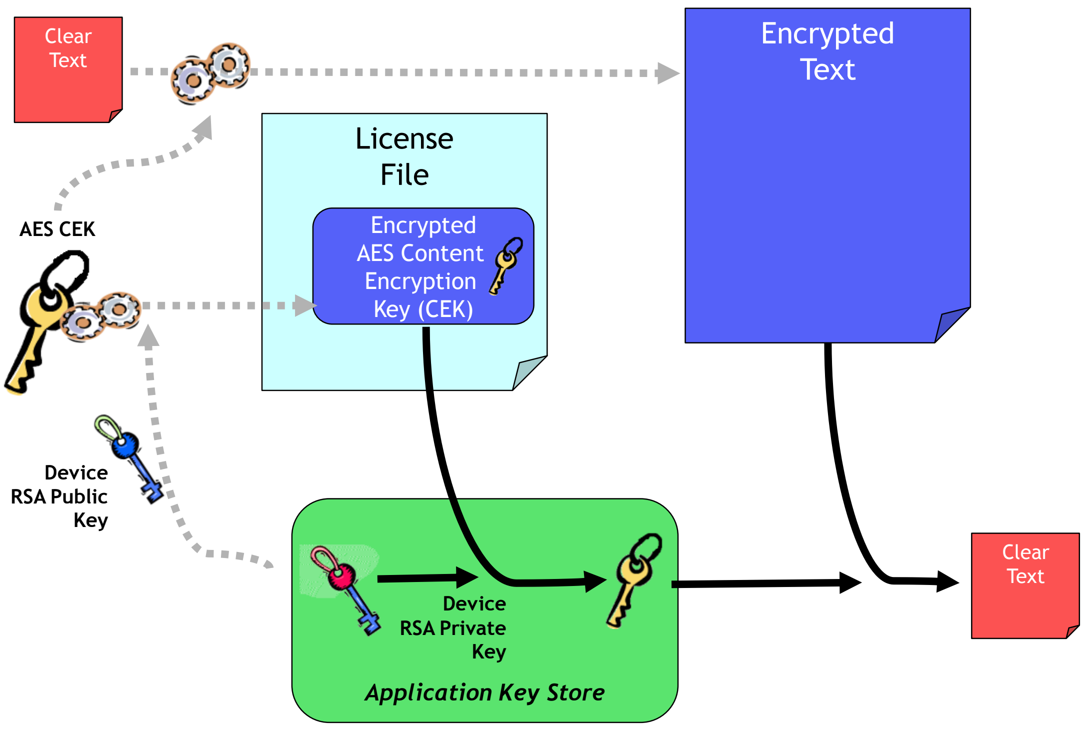


Figure A2-2: Crypto Example Processing.

The steps to protect the content:

- A 128-bit AES key is generated;
- The file is encrypted in AES-CBC using the generated key;
- The AES key is itself encrypted using a RSA public key;
- The AES encrypted key is stored in a file called a license;
- The license is decrypted using the RSA private key stored internally within the Trusted Foundations.

The following figure depicts the main components involved in this example:

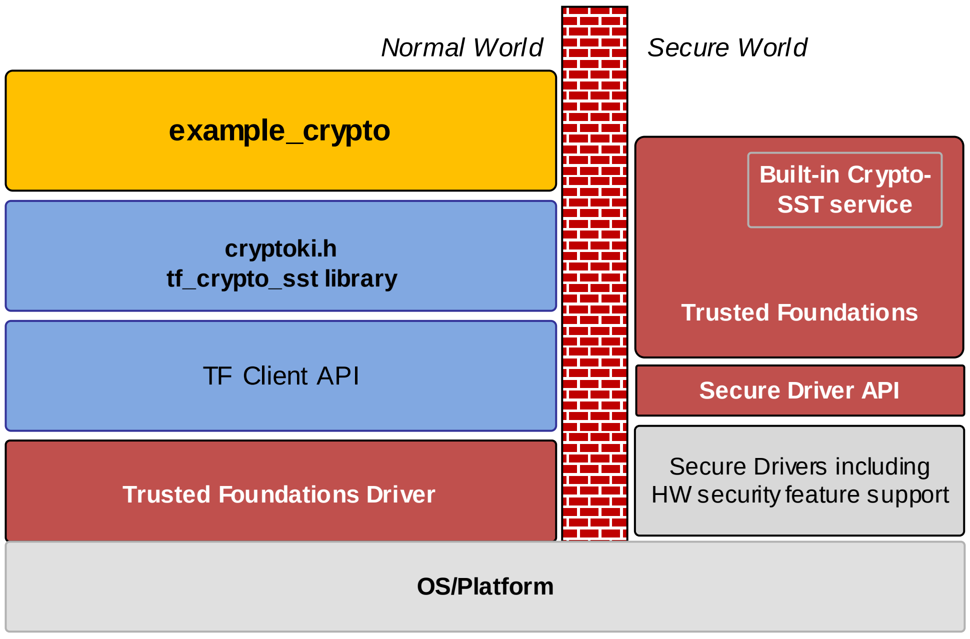


Figure A2-3: Crypto Example Architecture.

Sequence of Operations

The Sequence of Operations is the following:

- First, during an initialization step, the RSA private key used to decrypt licenses must be installed in the application key store within the Trusted Foundations In a real DRM scenario, this operation is typically performed once during the platform configuration (DRM Key Provisioning). For the sake of simplicity of this example, the RSA private key is imported in clear during this initialization step. The private key is stored permanently. That is, the initialization step must be performed only once. Other attempts will fail.
- Then the example application decrypts a given content using a given license through the following steps:
    - The license file is decrypted using the RSA private key.
    - The license file contains the AES key, which is installed temporarily in the application key store within the Trusted Foundations after it is decrypted.
    - The content file is decrypted using the AES key. The file is read and decrypted in blocks. This implementation uses in-place decryption to reduce the number of copy operations required when talking to the Trusted Foundations.
    - After the decryption of each block, the clear-text content of that block is displayed using `STDOUT`.

A2.3.2 User Guide

How to build the example

This example is located in the directory `examples/example_crypto` in the SDK.

In this example, only a Normal World component (the client application `example_crypto`) must be built.

Refer to section A2.1.1 to know how to build the example for a specific target.

Execution of a client request

First, start the Secure World.

Then execute the client application `example_crypto` using the following command line.

Usage:

```
example_crypto [options] <license_path> <content_path>
```

Decrypts contents of the content file with the license key.

Arguments:

- `license_path`: the path to a file containing the license.
- `content_path`: the path to a file containing the content.

Options:

- `-h`, `-help`: display this help
- `-slotID <slotId>`: the slot ID to use is `slotId`, specified as a decimal or a hexadecimal number. This parameter is optional. By default, the slot 1 (application-private slot) is used. This parameter can be used to select another slot, e.g., to access the token shared between all applications;
- `-init`: initialize the keystore used by this application. This should typically be performed once during the platform configuration, and must be done before the decryption of the content file.
- `-clean`: clean the keystore used by this application.

The first step is to initialize the keystore used by this application (using `example_crypto -init`). Once initialization has been completed, the application can then be run to decrypt content against a particular license (using `example_crypto <license path> <encrypted content path>`).

As decryption progresses, the resulting clear-text content is sent to `STDOUT`, and will display in the command window of the example. Note that decryption will proceed even if your license and content are not a correct pairing, and the corresponding "decrypted" content will still be scrambled as the key used is not the correct one.

Note that the initialization step can be merged with the first issue of the decryption command: `> example_crypto -init <license path> <encrypted content path>`

Example 1

Command:

```
>example_crypto.exe -init
```

Client trace:

```
Initialization succeed.
Decrypted content:
```

Example 2

Command:

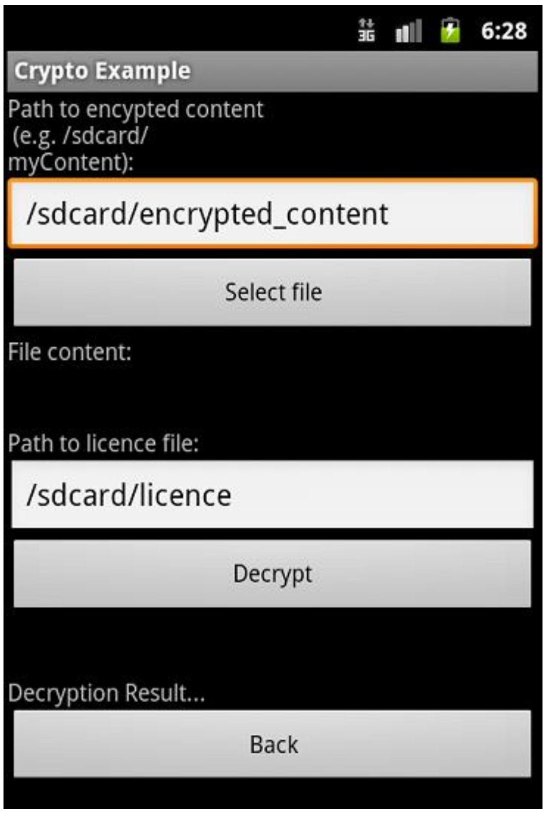
```
>example_crypto.exe -init license_2 encrypted_content_2
```

Client trace:

```
--- BEGIN ---
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate.
Rough winds do shake the darling buds of May,
And summer`s lease hath all too short a date.
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm`d;
And every fair from fair some time declines,
By chance, or nature`s changing course, untrimm`d;
But thy eternal summer shall not fade
Nor lose possession of that fair thou ow`st;
Nor shall Death brag thou wand`rest in his shade,
When in eternal lines to time thou grow`st:
So long as men can breath or eyes can see,
So long lives this, and this gives life to thee.
```

```
--- END ---
```

For the Android Functional Demonstrator, when using the APK version, a GUI is presented allowing the selection of the license path and encrypted content path from a drop-down list. We can also see the encrypted content as well as the decrypted result.


Crypto Example

Path to encypted content (e.g. /sdcard/ myContent):

/sdcard/encrypted_content

File content:

Path to licence file:

/sdcard/licence

Decryption Result...

A2.4 External Secure Storage Example

This section describes the External Secure Storage Example. It shows a simple usage of the External Secure Storage API, specified in Chapter B5.

A2.4.1 Overview

This is an example application that accesses the Secure Storage core service.

The following figure depicts the main components of the architecture interacting in this example:

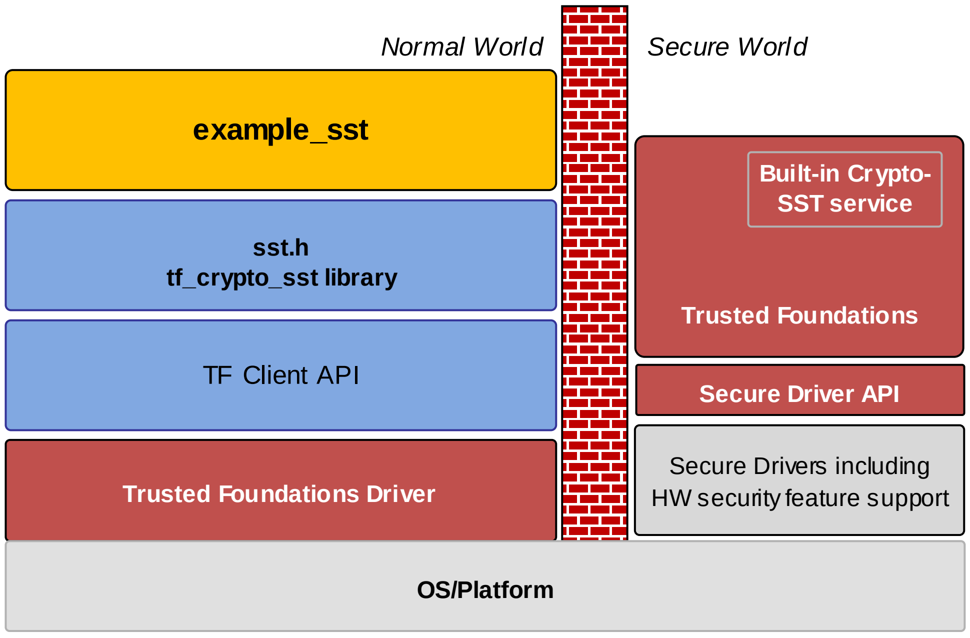


Figure A2-4: Secure Storage Example Architecture.

A2.4.2 User Guide

How to build the example

This example is located in the directory `examples/example_sst` of the SDK.

In this example, only a Normal World component (the client application `example_sst`) must be built.

Refer to section A2.1.1 to know how to build the example in a specific target.

Execution of a client request

First, start the Secure World.

Then start the example application:

- Open a command shell window, and navigate into the directory that contains the executable file `example_sst` (or `example_sst.exe`, depending on the target).
- Execute "`example_sst`" to run the secure storage client example.
- You should observe traces indicating operations results similar to the following:

```
Secure Storage Service Example Application
Copyright (c) 2005-2009 Trusted Logic S.A.

SST: Initialize the service :                               SST_SUCCESS

SST: Create a file :                                        SST_SUCCESS

SST: Write first data block :                               SST_SUCCESS
```

```
SST: Write second data block :                       SST_SUCCESS

SST: Checking the EOF :                              SST_SUCCESS

SST: Seeking to the start of the file :              SST_SUCCESS

SST: Read back first data block :                    SST_SUCCESS

SST: Seek to the start of second data block :        SST_SUCCESS

SST: Read back second data block :                   SST_SUCCESS

SST: Remove second data block by truncating :        SST_SUCCESS

SST: Get file pointer position :                     SST_SUCCESS

SST: Get file size :                                 SST_SUCCESS

SST: Remove the file :                               SST_SUCCESS

SST: Terminate the service :                         SST_SUCCESS

Example Summary : SUCCESS
```

A2.5 External Monotonic Counter Example

This section describes the External Monotonic Counter Example. It shows a simple usage of the External Monotonic Counter API, specified in Chapter B6.

A2.5.1 Overview

This is an example application that accesses the External Monotonic Counter API.

The following figure depicts the main components of the architecture interacting in this example:

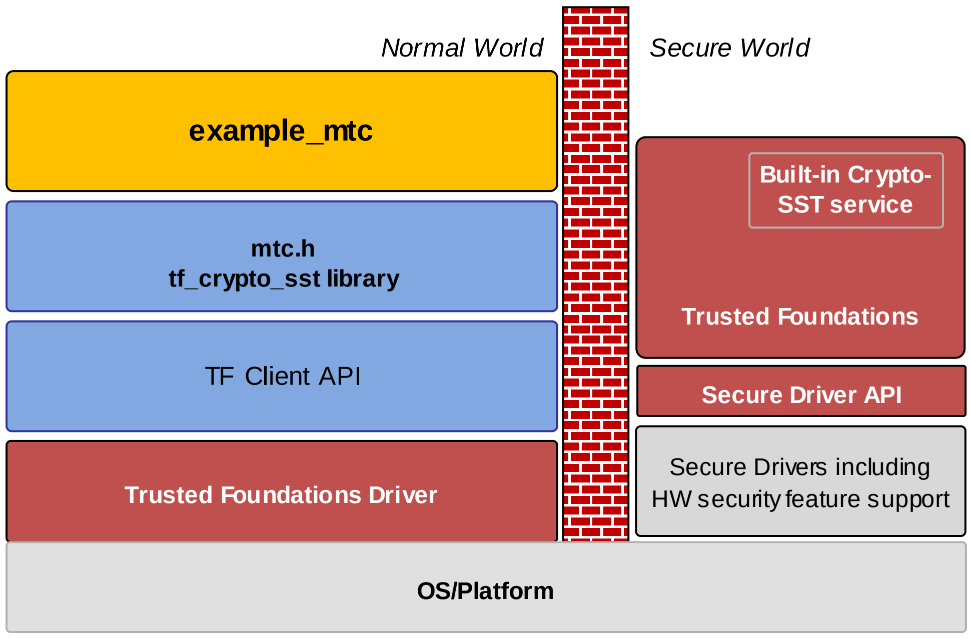


Figure A2-5: Monotonic Counter Example.

A2.5.2 User Guide

How to build the example

This example is located in the directory `examples/example_mtc` in the SDK.

In this example, only a Normal World component (the client application `example_mtc`) must be built.

Refer to section A2.1.1 to know how to build the example in a specific target.

Execution of a client request

First, start the Secure World.

Then start the example application:

- Open a command shell window, and navigate into the directory that contains the execution file `example_mtc` (or `example_mtc.exe`, depending on the target).
- Execute "`example_mtc`" to run the Monotonic Counter client example.
- You should observe traces indicating operations results, similar to the following:

```
Monotonic Counter Value is : 54081BAB14D1EF33
Monotonic Counter Value is : 54081BAB14D1EF34
```

A2.6 DRM EXAMPLE

This example describes the DRM Example. It shows a simple Secure Services using SSDI and the Internal Cryptographic API. See Chapter B2 for the specification of SSDI.

A2.6.1 OVERVIEW

This example includes a secure service written in C using the SSDI interface, a stub to implement a secure service-specific API over the TFAPI, and a client (i.e. security-demanding application) to test the functionalities of the secure service.

The service implements a very simple DRM agent to show the features of the TFAPI and SSDI. Actually, the functionality of this DRM agent is equivalent to the External Crypto Example. However, in the DRM Example, the Content Encryption Key is manipulated in the Secure World only and is not exposed in clear in the Normal World.

The following figure depicts the main components of the architecture interacting in this example:

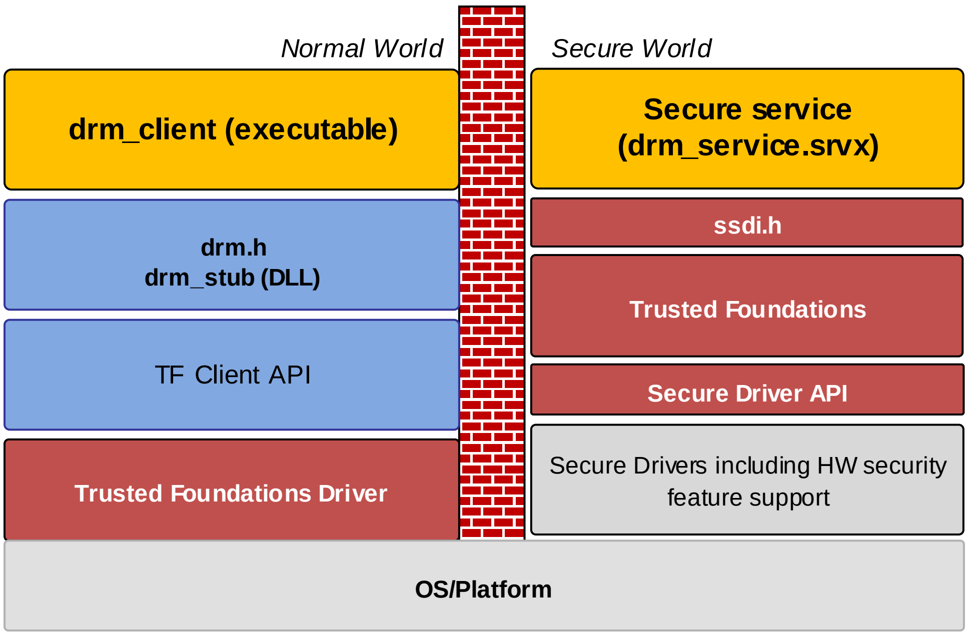


Figure A2-6: DRM Example Architecture.

Content

This example includes the following elements and files:

- *Secure Service*: The secure service source files are:
    - `drm_protocol.h`: command identifier definitions for the protocol between the Secure Service Stub and the Secure Service
    - `drm_service.c`: source of the secure service
    - `drm_service_init_data.h`: the device RSA private key
    - `drm.manifest.txt`: The service properties (meta-data)
- *Secure Service Stub*: The secure service stub source files are:
    - `drm_stub.c`: The source of the Secure Service Stub
    - `drm_stub.h`: The API exposed by the Secure Service Stub
- *Client*: The client source files are:
    - `drm_client.c`: The source of the client using the API of the Secure Service Stub
    - `drm_client.h`: Client header file
    - `drm_client.manifest.txt`: the manifest for client authentication

   - `data/`: A set of licenses and encrypted content to test the service

Functionalities

This example shows the following functionalities:

- how to implement a stub in order to offer a friendly interface to the drm client;
- how the service can restrict the access to some features (e.g. content decryption) to authenticated clients with specified properties in their manifest. See sectionB3.2.6 for more information about Client Authentication;
- how to use the Cryptoki Update Shortcut for decryption. See section B2.3.14.1 for more information about the Cryptoki Update Shortcut.

A2.6.2 USER GUIDE

How to build the example

This example is located in the directory `examples/example_drm` of the SDK.

In this example, both Normal World components (the client application `drm_client`, the secure service stub `drm_stub`) and Secure World components (the secure service `drm_service.srvx`) must be built.

Refer to section A2.1.1 to know how to build the example for a specific target.

Execution of a client request

First, start the Secure World, with the service `drm_service.srvx`.

Then execute the client application `drm_client` using the following command line.

Usage:

```
drm_client [options] <license_path> <content_path>
```

Arguments:

- *`license_path`*: the path to a file containing the license.
- *`content_path`*: the path to a file containing the content.

Options:

- `-h`, `-help`: display this help
- `-init`: initialize the drm root key. This should typically be performed once during the platform configuration, and must be done before the decryption of the content file.
- `-clean`: clean the keystore used by this application.

The first step is to initialize the drm root key (using `drm_client -init`). Once initialization has been completed, the client application can then be run to decrypt a content against a particular license (using `drm_client <license_path> <content_path>`).

As decryption progresses, the resulting clear-text content is sent to `STDOUT`, and will display in the command window of the example. Note that decryption will proceed even if your license and content are not a correct pairing, and the corresponding "decrypted" content will still be scrambled as the key used is not the correct one.

Note that the initialization step can be merged with the first issue of the decryption command:

```
> drm_client -init <license_path> <content_path>
```

Example 1

Command:

```
>drm_client.exe -init
```

Client trace:

```
  Initialization succeed.
```

```
Service trace:

 DRM_SAMPLE ----- SRVXCreate
 DRM_SAMPLE ----- SRVXOpenClientSession
 DRM_SAMPLE ----- CLIENT UUID: FA1B2859-396C-1E6C-2F9A-1FF9CB9CFC32
 DRM_SAMPLE ----- CLIENT PROPERTIES:
 DRM_SAMPLE -----  sm.client.login: 0x00000004
 DRM_SAMPLE ----- SRVXInvokeCommand
 DRM_SAMPLE ----- SRVXCloseClientSession
 DRM_SAMPLE ----- SRVXDestroy
```

Example 2

Command:

```
>drm_client.exe license_2 encrypted_content_2
```

Client trace:

```
Decrypted content:
--- BEGIN ---
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate.
Rough winds do shake the darling buds of May,
And summer`s lease hath all too short a date.
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm`d;
And every fair from fair some time declines,
By chance, or nature`s changing course, untrimm`d;
But thy eternal summer shall not fade
Nor lose possession of that fair thou ow`st;
Nor shall Death brag thou wand`rest in his shade,
When in eternal lines to time thou grow`st:
So long as men can breath or eyes can see,
So long lives this, and this gives life to thee.
--- END ---
```

Service trace:

```
DRM_SAMPLE ----- SRVXCreate
DRM_SAMPLE ----- SRVXOpenClientSession
DRM_SAMPLE ----- CLIENT UUID: 9937EEAA-C88E-4817-9218-2B8947D5B0E2
DRM_SAMPLE ----- CLIENT PROPERTIES:
DRM_SAMPLE -----  example_drm.allow_decrypt_content: true
DRM_SAMPLE -----  sm.client.exec.hash: mffERXKiC7WvJ0oV8hZEsVd2JzE=
DRM_SAMPLE -----  sm.client.login: 0x80000000
DRM_SAMPLE ----- SRVXCloseClientSession
DRM_SAMPLE ----- SRVXDestroy
```

A2.7 CRYPTOCATALOGUE EXAMPLE

This section describes the Crypto Catalogue Example. This example shows how to call the Cryptographic API, specified in Chapter B4, from the Secure Service to perform typical cryptographic operations.

A2.7.1 OVERVIEW

This example is a catalogue of most cryptographic mechanisms:

- key creation/generation/destruction,
- CKO_DATA object creation,
- digest (SHA1),
- encryption/decryption (RSA PKCS, RSA OAEP, AES, DES, DES3),
- signature/verification (RSA PKCS, RSA PSS, DSA),
- random number generation,
- key agreement (DH).

Mechanisms are performed with single-part and/or multi-part computation. A lot of comments give indications about the mechanisms.

This code is meant to be used as a starting point for developers who want to use the Cryptographic API. The example uses a Secure Service calling the Internal Cryptographic API but the same code can be adapted to use the External Cryptographic API as well.

A2.7.2 USER GUIDE

How to build the example

This example is located in the directory `examples/example_cryptocatalogue` of the SDK.

In this example, both Normal World components (the client application `cryptocatalogue_client`) and Secure World components (the secure service `cryptocatalogue_service.srvx`) must be built.

Refer to section A2.1.1 to know how to build the example for a specific target.

Execution of a client request

First, start the Secure World, with the service `cryptocatalogue_service.srvx`.

Then execute the client application `cryptocatalogue_client` using the following command line.

Usage:

```
>cryptocatalogue_client.exe
```

Service trace:

```
CRYPTOCATALOGUE  SRVXCreate
CRYPTOCATALOGUE  SRVXOpenClientSession
CRYPTOCATALOGUE  SRVXInvokeCommand
CRYPTOCATALOGUE  Start cryptoCatalogueService()
CRYPTOCATALOGUE  exampleDigestMechanism:
CRYPTOCATALOGUE   -  SHA1 digest, one stage (then comparison)
CRYPTOCATALOGUE   -  SHA1 digest, multi stages (then comparison)
CRYPTOCATALOGUE  exampleRsaPkcsMechanismSignVerify:
CRYPTOCATALOGUE   -  Import a RSA key pair
CRYPTOCATALOGUE   -  RSA PKCS SHA1 signature, one stage (then comparison)
CRYPTOCATALOGUE   -  RSA PKCS SHA1 signature, multi stages (then comparison)
CRYPTOCATALOGUE   -  RSA PKCS SHA1 verification, one stage
CRYPTOCATALOGUE   -  RSA PKCS SHA1 verification, multi stages
CRYPTOCATALOGUE   -  Destroy RSA key pair
CRYPTOCATALOGUE  exampleRsaPssMechanism:
CRYPTOCATALOGUE   -  Import a RSA key pair
CRYPTOCATALOGUE   -  RSA PSS SHA1 signature, one stage (then verification)
CRYPTOCATALOGUE   -  RSA PSS SHA1 signature, multi stages (then verification)
CRYPTOCATALOGUE   -  RSA PSS SHA1 verification, one stage
CRYPTOCATALOGUE   -  RSA PSS SHA1 verification, multi stages
CRYPTOCATALOGUE   -  SHA1 digest + RSAPSS signature, one stage (then verification, two dif
CRYPTOCATALOGUE  ferent ways)
CRYPTOCATALOGUE   -  Destroy RSA key pair
CRYPTOCATALOGUE  exampleDsaMechanism:
CRYPTOCATALOGUE   -  Import a DSA key pair
CRYPTOCATALOGUE   -  DSA SHA1 signature, one stage (then verification)
CRYPTOCATALOGUE   -  DSA SHA1 signature, multi stages (then verification)
CRYPTOCATALOGUE   -  DSA SHA1 verification, one stage
CRYPTOCATALOGUE   -  DSA SHA1 verification, multi stages
CRYPTOCATALOGUE   -  SHA1 digest + DSA signature, one stage (then verification, two differe
CRYPTOCATALOGUE  nt ways)
CRYPTOCATALOGUE   -  Destroy DSA key pair
CRYPTOCATALOGUE  exampleDsaKeyGeneration:
CRYPTOCATALOGUE   -  Generate a DSA key pair
CRYPTOCATALOGUE   -  DSA SHA1 signature, multi stages (then verification)
CRYPTOCATALOGUE   -  Destroy DSA key pair
CRYPTOCATALOGUE  exampleRsaPkcsMechanismEncryptDecrypt:
CRYPTOCATALOGUE   -  Import a RSA key pair
CRYPTOCATALOGUE   -  RSA PKCS encryption, one stage (then decryption)
CRYPTOCATALOGUE   -  RSA PKCS decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  Destroy RSA key pair
CRYPTOCATALOGUE  exampleRsaKeyGeneration:
CRYPTOCATALOGUE   -  Generate a RSA key pair
CRYPTOCATALOGUE   -  RSA PKCS encryption, one stage (then decryption)
CRYPTOCATALOGUE   -  Destroy RSA key pair
CRYPTOCATALOGUE  exampleRsaOaepMechanism:
CRYPTOCATALOGUE   -  Import a RSA key pair
CRYPTOCATALOGUE   -  RSA OAEP SHA1 encryption, one stage (then decryption)
CRYPTOCATALOGUE   -  RSA OAEP SHA1 decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  Destroy RSA key pair
CRYPTOCATALOGUE  exampleAesMechanism:
CRYPTOCATALOGUE   -  Import a AES key 128 bits
CRYPTOCATALOGUE   -  Import a AES key 192 bits
CRYPTOCATALOGUE   -  Import a AES key 256 bits
CRYPTOCATALOGUE   -  AES CTR 128 encryption, one stage (then comparison)
CRYPTOCATALOGUE   -  AES CTR 128 encryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  AES CTR 128 decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  AES CTR 128 decryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  AES ECB 192 encryption, one stage (then comparison)
CRYPTOCATALOGUE   -  AES ECB 192 encryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  AES ECB 192 decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  AES ECB 192 decryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  AES CBC 256 encryption, one stage (then comparison)
CRYPTOCATALOGUE   -  AES CBC 256 encryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  AES CBC 256 decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  AES CBC 256 decryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  Destroy AES keys
CRYPTOCATALOGUE  exampleAesKeyGeneration:
CRYPTOCATALOGUE   -  Generate a AES key 128
CRYPTOCATALOGUE   -  AES 128 ECB encryption, one stage (then decryption)
CRYPTOCATALOGUE   -  Destroy AES key
CRYPTOCATALOGUE  exampleDesMechanism:
CRYPTOCATALOGUE   -  Import a DES key (64 bits)
CRYPTOCATALOGUE   -  Import a DES2 key (128 bits)
CRYPTOCATALOGUE   -  Import a DES3 key (192 bits)
CRYPTOCATALOGUE   -  DES ECB encryption, one stage (then comparison)
CRYPTOCATALOGUE   -  DES ECB encryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES ECB decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  DES ECB decryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES CBC encryption, one stage (then comparison)
CRYPTOCATALOGUE   -  DES CBC encryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES CBC decryption, one stage (then comparison)
CRYPTOCATALOGUE   -  DES CBC decryption, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES3 ECB encryption, DES3 key, one stage (then comparison)
CRYPTOCATALOGUE   -  DES3 ECB encryption, DES3 key, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES3 ECB decryption, DES3 key, one stage (then comparison)
CRYPTOCATALOGUE   -  DES3 ECB decryption, DES3 key, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES3 CBC encryption, DES2 key, one stage (then comparison)
CRYPTOCATALOGUE   -  DES3 CBC encryption, DES2 key, multi stages (then comparison)
CRYPTOCATALOGUE   -  DES3 CBC decryption, DES2 key, one stage (then comparison)
CRYPTOCATALOGUE   -  DES3 CBC decryption, DES2 key, multi stages (then comparison)
CRYPTOCATALOGUE   -  Destroy DES/DES2/DES3 keys
CRYPTOCATALOGUE  exampleDesKeyGeneration:
CRYPTOCATALOGUE   -  Generate a DES2 key
CRYPTOCATALOGUE   -  DES3 ECB encryption, DES2 key, multi stages (then decryption)
CRYPTOCATALOGUE   -  Destroy DES2 key
CRYPTOCATALOGUE  exampleRandomMechanism:
CRYPTOCATALOGUE   -  Generate a random
CRYPTOCATALOGUE   -  Use it to create a AES 128 bits key
CRYPTOCATALOGUE  exampleDhMechanism1:
CRYPTOCATALOGUE   -  Import DH private key
CRYPTOCATALOGUE   -  Derive secret key
CRYPTOCATALOGUE   -  Retrieve secret key value (then comparison)
CRYPTOCATALOGUE   -  Destroy keys
CRYPTOCATALOGUE  exampleDhMechanism2:
CRYPTOCATALOGUE   -  Generate DH key pair
CRYPTOCATALOGUE   -  Derive secret key
CRYPTOCATALOGUE   -  Try to retrieve secret key value (must fail)
CRYPTOCATALOGUE   -  Destroy keys
CRYPTOCATALOGUE  exampleDhMechanism3:
CRYPTOCATALOGUE   -  Generate DH key pair
CRYPTOCATALOGUE   -  Derive secret key
CRYPTOCATALOGUE   -  Retrieve secret key value
CRYPTOCATALOGUE   -  Use it for AES encryption (then decryption)
CRYPTOCATALOGUE   -  Destroy keys
CRYPTOCATALOGUE  exampleCkoDataObject:
CRYPTOCATALOGUE   -  Create a CKO_DATA object from data (128 bytes)
CRYPTOCATALOGUE   -  Retrieve the object value
CRYPTOCATALOGUE   -  Compare with the data value
CRYPTOCATALOGUE   -  Destroy the object
CRYPTOCATALOGUE  cryptoCatalogueService() was successfully performed.
CRYPTOCATALOGUE  SRVXCloseClientSession
CRYPTOCATALOGUE  SRVXDestroy
```

A2.8 CERTSTORE EXAMPLE

This section descr ibes the Certificate Store Example. This example shows a secure service calling the SFile API.

A2.8.1 OVERVIEW

The secure service in this example implements a simple Certificate Store service. The `SFile` is used to protect the integrity of the stored certificates. Client Authentication is used to control the modifications of the certificates.

The following commands are implemented by the service:

- install a X509 certificate in the certstore. This requires the client to be authenticated. The certificate is parsed, stored, and an identifier is returned to the client;
- parse the certificate to retrieve the Distinguished Name (DN) field;
- delete a certificate from its index (requires the client to be authenticated);
- delete all the installed certificate (requires the client to be authenticated);
- list all the certificates indexes;
- retrieve the certificate to a client;
- verify the signature of a certificate against if one certificate is issued by the other one.

This example shows the following functionalities:

- how to use the `SFile` API;
- how to retrieve the client UUID and client properties (displayed at each client session opening);
- how to control access to sensitive operations (like installation or deletetion of certificate) and restrict the access depending on the client login type (authenticated or not) and its properties (specifying whether the client has management rights);
- how to encode/decode different types of objects (32-bit words, byte arrays) in both normal word (using TFAPI) and secure world (using SSDI API).

A2.8.2 USER GUIDE

This example is located in the directory `examples\example_certstore` in the SDK.

How to build the example

In this example, both Normal World components (the client application `certstore_client` and Secure World components (the secure service `certstore_service.srvx`) must be built.

Refer to section A2.1.1 to know how to build the example in a specific target.

Execution of a client request

First, start the Secure World, with the service `certstore_service.srvx`.

Then execute the client application `certstore_client` using the following command line.

Usage:

`certstore_client <`*`command`*`>`

Supported commands are:

- `help` or `-help`: Display this help
- `installCert <`*`certificate path`*`>`: Install a certificate. This option must be followed by a certificate path. The service returns a certificate identifier, and the Distinguished Name (DN) of the certificate;
- `deleteCert <certificate identifier>`: Delete a certificate. This option must be followed by a certificate identifier;
- `listAll`: List all certificates already installed. (returns all certificates identifiers);

- `readCertDN` *<certificate identifier>*: Return the DN of a certificate. The service returns the certificate DN formatted as a list of strings.
- `getCert` *<certificate identifier>* `[`*<output_path>*`]`: Get a certificate. This option must be followed by a certificate identifier. The service returns the whole certificate and save it in the current path
- `verifyCert` *<certificate identifier>* *<issuer_certificate identifier>*:Verify that the first certificate is issued by the second one.
- `deleteAll`: Delete all certificates.

Remark 1:

- *<certificate path>*: the path to a file containing a certificate at DER format.
- *<certificate identifier>*: the identifier of a certificate already installed.
- *<output_path>*: the path to a file for storing the certificate.

Remark 2:

The command `verifyCert` verifies the signature of the certificate signature against the issuer's public key.

This example supports only the `sha1WithRSAEncryption` signature algorithm (`OID 1:2:840:113549:1:1:5`).

Example 1

Command:

```
> certstore_client installCert ..\..\..\data\example_certstore.der
```

Client trace:

```
  Trusted Logic CERTSTORE Example Application
  Start clientInstallCertificate().
  The Certificate is successfully installed.
  CERTIFICATE IDENTIFIER: 0x00000000.
  DISTINGUISHED NAME:
     C  : FR
     ST : Yvelines
     L  : Versailles
     O  : Trusted Logic S.A.
     OU : BU PCD
     CN : This is a test certificate
     E  : certstore_example@trusted-logic.com
```

Service trace:

```
  CERTSTORE EXAMP  SRVXCreate
  CERTSTORE EXAMP  SRVXOpenClientSession
  CERTSTORE EXAMP  CLIENT UUID: 7EF2A52B-9041-4DE0-9C88-B0A0EB6BBC35
  CERTSTORE EXAMP  CLIENT PROPERTIES:
  CERTSTORE EXAMP   example_certstore.allow_manager_operations: true
  CERTSTORE EXAMP   sm.client.exec.hash:fzf4z/2+9vpMs5tqclGSYLoTVc4=
  CERTSTORE EXAMP   sm.client.login: 0x8000000
  CERTSTORE EXAMP  SRVXInvokeCommand
  CERTSTORE EXAMP  Start serviceInstallCertificate().
  CERTSTORE EXAMP  CERTIFICATE IDENTIFIER : (0x00000001).
  CERTSTORE EXAMP  DISTINGUISHED NAME:
  CERTSTORE EXAMP     C : FR
  CERTSTORE EXAMP     ST: Yvelines
  CERTSTORE EXAMP     L : Versailles
  CERTSTORE EXAMP     O : Trusted Logic S.A.
  CERTSTORE EXAMP     OU: BU PCD
  CERTSTORE EXAMP     CN: This is a test certificate
  CERTSTORE EXAMP     E : certstore_example@trusted-logic.com
  CERTSTORE EXAMP  SRVXCloseClientSession
  CERTSTORE EXAMP  SRVXDestroy
```

Command:

```
> certstore_client deleteCert  0x00000000
```

Client trace:

```
  Trusted Logic CERTSTORE Example Application
  Start clientDeleteCertificate().
  The Certificate 0x00000001 is successfully removed.
```

Service trace:

```
  CERTSTORE EXAMP  SRVXOpenClientSession
  CERTSTORE EXAMP  CLIENT UUID: 7EF2A52B-9041-4DE0-9C88-B0A0EB6BBC35
  CERTSTORE EXAMP  CLIENT PROPERTIES:
  CERTSTORE EXAMP   example_certstore.allow_manager_operations: true
  CERTSTORE EXAMP   sm.client.exec.hash: fzf4z/2+9vpMs5tqclGSYLoTVc4=
  CERTSTORE EXAMP   sm.client.login: 0x80000000
  CERTSTORE EXAMP  SRVXInvokeCommand
  CERTSTORE EXAMP  Start serviceDeleteCertificate().
  CERTSTORE EXAMP  SRVXCloseClientSession
```

Example 2

If the client is not signed, installing a certificate will fail. To test this scenario, remove the signature file `certstore_client.exe.ssig` and retry::

```
> move certstore_client.exe.ssig certstore_client.exe.ssig.bak
> certstore_client installCert ..\..\..\data\example_certstore.der
```

Client trace:

```
  Trusted Logic CERTSTORE Example Application
  Start clientInstallCertificate().
  -        ERROR: Could not perform invoke command (ffff1000).
  -        ERROR: Access denied.
```

Service trace:

```
  CERTSTORE EXAMP  SRVXOpenClientSession
  CERTSTORE EXAMP  CLIENT UUID: C52508A6-8FAE-3EB3-5608-77D94FD5C24E
  CERTSTORE EXAMP  CLIENT PROPERTIES:
  CERTSTORE EXAMP   sm.client.login: 0x00000001
  CERTSTORE EXAMP  SRVXInvokeCommand
  CERTSTORE EXAMP  Start serviceInstallCertificate().
  CERTSTORE EXAMP  ERROR: The client should have manager rights and be
  authenticated.
  CERTSTORE EXAMP  SRVXCloseClientSession
```

A2.9 SECURE DATE EXAMPLE

This section describes the Secure Date Example. This shows how to use the `SDate` API.

A2.9.1 OVERVIEW

The main purpose of the example is to show that each service has its own date and that changing a service's date does not affect the date of the other services.

A2.9.2 USER GUIDE

How to build the example

This example is located in the directory `examples/example_sdate` of the SDK. It includes a Normal World component and **two** Secure Services.

Refer to section A2.1.1 to know how to build the example for a specific target.

Example description

The client opens a session with the first service:

- the date of this services is retrieved and then set to a different date.

Service Trace:

```
SERVICE DATE A - Current date was not set.
SERVICE DATE A - Date was synchronizedto local device time.
SERVICE DATE A - Service Date : 2010/12/20 - 17:29:24
SERVICE DATE A - Date was changed to: 1999/1/30 - 11:12:13
SERVICE DATE A - Service Date : 1999/1/30 - 11:12:13
```

The client opens a session with the second service:

- the date of the second service is retrieved.

Service Trace:

```
SERVICE DATE B - Current date was not set.
SERVICE DATE B - Date was synchronized to local device time.
SERVICE DATE B - Service Date : 2010/12/20 - 17:29:24
```

Note that the date of the second service has not been affected by the change of date in the first service.

Part B

SPECIFICATIONS OF THE DEVELOPER APIs V3.0

This part contains the complete reference of all the Developer APIs, version 3.x, also known as the "Advanced Profile". It contains the following chapters:

- Chapter B1, "*Common Definitions*": general definitions used by all the other chapters
- Chapter B2, "*Secure Service Development Interface*": API used for the development of Secure Services;
- Chapter B3, "*Trusted Foundations Client API*": API used to access Secure Services from the Normal World;
- Chapter B4, "*Trusted Foundations Cryptographic API*": the Cryptographic API is used both from Secure Services executing in the Secure World and from Normal World Applications;
- Chapter B5, "*External Secure Storage API*": Normal World API for Secure Storage;
- Chapter B6, "*External Monotonic Counter API*": Normal World API for Monotonic Counter;
- Chapter B7, "*Service Manager Protocol*": the protocol for Normal World or Secure World clients to access the Service Manager, to inspect the installed services and to manage them.

Chapter B1 COMMON DEFINITIONS

This chapter contains definitions used throughout the Developer API specifications.

B1.1 Common Types

B1.1.1 Basic Types

These specifications make use of the integer and boolean C types as defined in the C99 standard (ISO/IEC 9899:1999). The following basic types are used:

- `uint32_t`: unsigned 32-bit integer;
- `int32_t`: signed 32-bit integer;
- `uint16_t`: unsigned 16-bit integer;
- `int16_t`: signed 16-bit integer;
- `uint8_t`: unsigned 8-bit integer;
- `int8_t`: signed 8-bit integer;
- `bool`: Boolean type with the values `true` and `false`;
- `char`: a character. This type is used to denote a byte in a UTF-8 encoded Unicode string.

B1.1.2 S_HANDLE, SM_HANDLE

```
typedef uint32_t S_HANDLE;
```

This is the type used to contain opaque handles returned by the APIs.

For compatibility reasons, the following alias of this type is also defined:

```
typedef S_HANDLE SM_HANDLE;
```

B1.1.3 S_RESULT, TEEC_RESULT, SM_ERROR

```
typedef uint32_t S_RESULT;
```

This is the type used for return codes from the APIs.

For compatibility reasons, two aliases of this type are also defined:

```
typedef S_RESULT TEEC_RESULT;
typedef S_RESULT SM_ERROR;
```

B1.1.4 S_UUID, TEEC_UUID, SM_UUID

```
typedef struct S_UUID
{
   uint32_t time_low;
   uint16_t time_mid;
   uint16_t time_hi_and_version;
   uint8_t  clock_seq_and_node[8];
}
S_UUID;
```

The Universally Unique Resource Identifier type as defined in **[RFC4122]**. This type is used to identify services and clients.

UUIDs can be directly hard-coded in the service code. For example, the UUID `79B77788-9789-4a7a-A2BE-B60155EEF5F3` can be hard-coded using the following code:

```
static const S_UUID myUUID =
{
    0x79b77788, 0x9789, 0x4a7a,
    { 0xa2, 0xbe, 0xb6, 0x1, 0x55, 0xee, 0xf5, 0xf3 }
};
```

For compatibility reasons, two aliases of this type are also defined:

```
typedef S_UUID TEEC_UUID;
typedef S_UUID SM_UUID;
```

B1.2 COMMON ERROR CODES

The following error codes are used throughout the APIs.

For compatibility reasons, some aliases are available.

Note that these error codes are *not used* in the Cryptographic API specified in Chapter B4. The Cryptographic API makes use of a different set of error codes, defined in section B4.5.1.

Table B1-1 Common Error Codes

Constant Names and Aliases	Value
S_SUCCESS, SM_SUCCESS, SST_SUCCESS, TEEC_SUCCESS	0x00000000
S_ERROR_GENERIC, SM_ERROR_GENERIC, SST_ERROR_GENERIC, TEEC_ERROR_GENERIC	0xFFFF0000
S_ERROR_ACCESS_DENIED, SM_ERROR_ACCESS_DENIED, SST_ERROR_ACCESS_DENIED, TEEC_ERROR_ACCESS_DENIED	0xFFFF0001
S_ERROR_CANCEL, SM_ERROR_CANCEL, TEEC_ERROR_CANCEL	0xFFFF0002
S_ERROR_ACCESS_CONFLICT, SM_ERROR_EXCLUSIVE_ACCESS, SST_ERROR_ACCESS_CONFLICT, TEEC_ERROR_ACCESS_CONFLICT	0xFFFF0003
S_ERROR_EXCESS_DATA, SM_ERROR_EXCESS_DATA, TEEC_ERROR_EXCESS_DATA	0xFFFF0004
S_ERROR_BAD_FORMAT, SM_ERROR_FORMAT, TEEC_ERROR_BAD_FORMAT	0xFFFF0005
S_ERROR_BAD_PARAMETERS, SM_ERROR_ILLEGAL_ARGUMENT, SST_ERROR_BAD_PARAMETERS, TEEC_ERROR_BAD_PARAMETERS	0xFFFF0006
S_ERROR_BAD_STATE, SM_ERROR_ILLEGAL_STATE, TEEC_ERROR_BAD_STATE	0xFFFF0007
S_ERROR_ITEM_NOT_FOUND, SM_ERROR_ITEM_NOT_FOUND, SST_ERROR_ITEM_NOT_FOUND, TEEC_ERROR_ITEM_NOT_FOUND	0xFFFF0008
S_ERROR_NOT_IMPLEMENTED, SM_ERROR_NOT_IMPLEMENTED, TEEC_ERROR_NOT_IMPLEMENTED	0xFFFF0009
S_ERROR_NOT_SUPPORTED, SM_ERROR_NOT_SUPPORTED, TEEC_ERROR_NOT_SUPPORTED	0xFFFF000A
S_ERROR_NO_DATA, SM_ERROR_NO_DATA, TEEC_ERROR_NO_DATA	0xFFFF000B
S_ERROR_OUT_OF_MEMORY, SM_ERROR_OUT_OF_MEMORY, SST_ERROR_OUT_OF_MEMORY, TEEC_ERROR_OUT_OF_MEMORY	0xFFFF000C
S_ERROR_BUSY, SM_ERROR_BUSY, TEEC_ERROR_BUSY	0xFFFF000D
S_ERROR_COMMUNICATION, SM_ERROR_COMMUNICATION, TEEC_ERROR_COMMUNICATION	0xFFFF000E
S_ERROR_SECURITY, SM_ERROR_SECURITY, TEEC_ERROR_SECURITY	0xFFFF000F
S_ERROR_SHORT_BUFFER, SM_ERROR_SHORT_BUFFER, TEEC_ERROR_SHORT_BUFFER	0xFFFF0010
SM_ERROR_ASYNC_OPERATIONS_NOT_SUPPORTED	0xFFFF0011
S_ERROR_SERVICE, SM_ERROR_SERVICE	0xFFFF1000
S_PENDING, SM_PENDING	0xFFFF2000
S_ERROR_TIMEOUT	0xFFFF3001
TEEC_ERROR_OS	0xFFFF3002
S_ERROR_OVERFLOW SST_ERROR_OVERFLOW	0xFFFF300F
S_ERROR_TARGET_DEAD, SM_ERROR_TARGET_DEAD, TEEC_ERROR_TARGET_DEAD	0xFFFF3024
S_ERROR_STORAGE_NO_SPACE, SST_ERROR_NO_SPACE	0xFFFF3041
S_ERROR_STORAGE_CORRUPTED, SST_ERROR_CORRUPTED	0xFFFF3045
S_ERROR_STORAGE_UNREACHABLE	0xFFFF3046

Chapter B2 SECURE SERVICE DEVELOPMENT INTERFACE

This chapter is the specification of the Secure Service Development Interface (SSDI).

B2.1 About SSDI

This chapter is the specification for the Secure Service Development Interface (SSDI).

SSDI allows service developers to program secure services within the Trusted Foundations using the C language. Secure services are typically accessed through the TF Client API from applications in the Normal World, although they may also be accessed by other services in the Secure World.

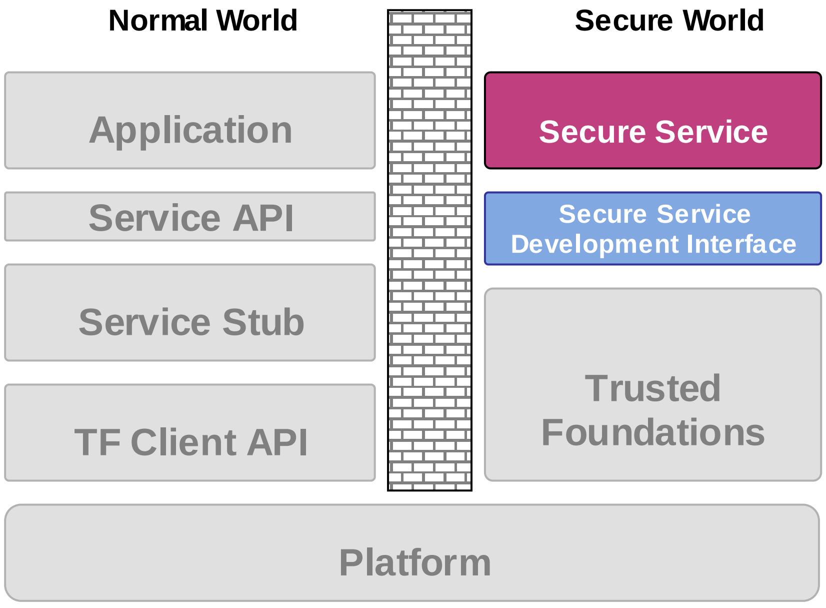


Figure B2-1: Secure Services and SSDI.

This chapter is organized as follows:

- Section B2.2, "*Using SSDI*" provides an overview of the various facilities offered by the SSDI;
- Section B2.3, "*SSDI Reference*" is the complete reference of all SSDI.

B2.2 Using SSDI

B2.2.1 Developing Secure Services

Services are developed using the C language. The way secure services are compiled, linked, and installed in the Trusted Foundations is implementation-dependent. Refer to Chapter A1 for more information.

The following C functionality is not directly supported by the service model:

- Writable variables with global scope.
- Writable static variables with global or function scope.

Read-only static data with global or function scope is allowed. This is achieved by typing the data with the `static` and `const` type qualifiers. For example:

```
static const uint8_t pPublicKey_e[] = { 0x01, 0x00, 0x01 };
```

As an alternative to writable global data there are functions to register and access an instance variable (see functions `SInstanceSetData` and `SInstanceGetData`) which are accessible from any thread within the instance. Typically this instance variable will be used to hold a pointer to a service-defined memory block containing any writable data that needs global scope, or writable static data that needs function scope.

In addition to this instance variable, each client session may also set a "session context" variable when the session is opened. This session context variable is passed back into each subsequent operation entry point within that session, allowing allocation and management of memory for session local global or static data.

B2.2.2 Secure Services

A Secure Service is a security-dedicated program that runs within the Secure World.

Each service is globally identified by a Universally Unique Identifier (UUID) as specified in **[RFC4122]**. The service identifier and other service properties are set by the programmer in the service manifest.

A service is command-oriented. Clients access a service by opening a session with the service and sending commands within the session. When the service receives a command, it parses any message associated with the command, performs any required processing and then sends a response back to the client.

When a client creates a session with a service it connects to an instance of that service. An *instance* of a service is a single process in the Secure World, with a memory space that is logically separated from all other processes. Each command issued by the client then exists within a single session connected to an instance.

A service may support multiple concurrent instances, sessions, and commands, depending on its configuration:

- Multiple instances: a service may allow a single instance which supports multiple sessions, or a new instance may be created for each new session;
- Multiple sessions: a single service instance may support multiple concurrent sessions. Note that a multi-instance service will only ever allow one session per instance.

There are therefore three kinds of services:

- Mono-instance, multi-session services: by default the service is instantiated at boot time; a new session in the single instance is created for each client session. The instance is destroyed when the system shuts down.
- Mono-instance, mono-session services: by default the service is instantiated at boot time; and will accept a single session at any one time. An attempt to create a second concurrent client session is rejected.

- Multi-instance services: A new instance is created for each client session that connects. Therefore a single instance of this service only ever supports one client session, and the multi-session property is ignored.

Additionally, there is a multi-command property that controls the number of simultaneous commands that you can have in a single client session. If the service supports only a single command at a time, then the system queues the other commands until the pending command has completed.

Finally, there is a lazy instancing property that is used to control when mono-instance services are instantiated. By default they are instantiated at boot time, and closed when the platform shuts down. When a service is configured to be lazy instancing, the new instance is created when the first client session connects, and is destroyed when the last client session disconnects.

The configuration of these behaviors is expressed through service properties, written in a manifest by the developer of the service. Service properties are used to configure the system, they can be accessed at runtime by the service itself, and some of the service properties are also visible from the clients of the service. See section B2.3.6.1 for more information about service properties.

Services must implement a Service Provider Interface called the SRVX interface and can make use of an Application Programming Interface (API) to access the facilities in the System, as shown in the following figure:

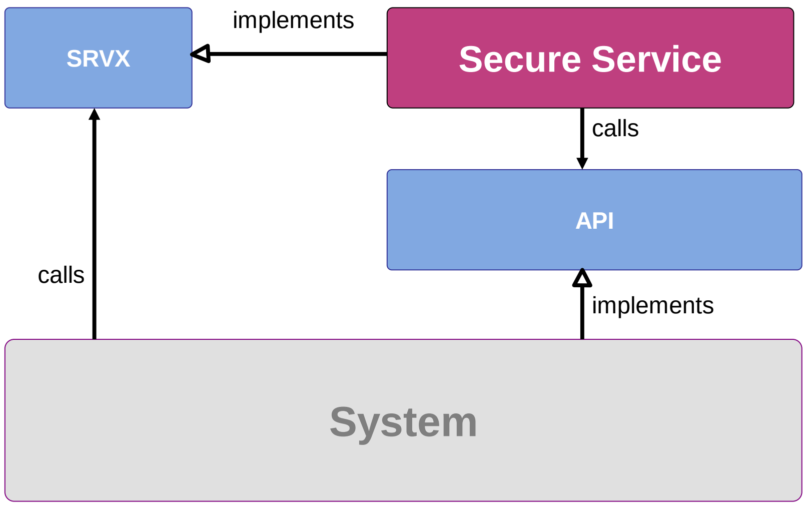


Figure B2-2: Secure Service SPI and API.

Note that, in this chapter, the term “System” denotes the product implementation as seen from the point of view of a service.

B2.2.3 Clients and Client Sessions

A service is typically accessed through the TF Client API. Normal World clients use this API to connect to a service and to invoke commands of the service. Refer to Chapter B3, “*Trusted Foundations Client API*” for more details on the TF Client API. It is also possible for a service to act as a client of another service, using the internal `SXControl` API to manage session and connections (see section B2.2.13).

When a client connects to a service, the system associates the session with a client identity, which is a UUID, and client properties.

Services can retrieve the identity and properties of their client by calling one of the session functions (see section B2.3.7, “*Session Functions*”). Most client properties are defined by the client application itself, but the system itself assigns the value of the `sm.client.login` property. The value of this property depends on the login method used by the client to connect to the service. It contains an integer which can be assigned one of the following values:

- `S_LOGIN_PUBLIC`: the client is in the Normal World and is neither identified, nor authenticated. The client has no identity and the UUID returned by `SSessionGetClientID` is the Nil UUID defined in **[RFC4122]**;

- `S_LOGIN_PRIVILEGED`: the client is a privileged application. The exact meaning of "privileged" depends on the Operating System. The client identity is the nil UUID;
- `S_LOGIN_APPLICATION`: the client application has been identified by the Normal World Operating System independently of the identity of the user executing the application. The nature of this identification and the corresponding UUID is OS-specific;
- `S_LOGIN_USER`: the client application has been identified by the Normal World Operating System and the client UUID reflects the actual user that runs the calling application independently of the actual application;
- `S_LOGIN_GROUP`: the client UUID reflects a group identity that is executing the calling application. The notion of group identity and the corresponding UUID is OS-specific
- `S_LOGIN_APPLICATION_USER`: the client UUID identifies both the calling application and the user that is executing it;
- `S_LOGIN_APPLICATION_GROUP`: the client UUID identifies both the calling application and a group that is executing it;
- `S_LOGIN_AUTHENTICATION`: the client and its client properties have been authenticated against a trusted signer certificate. The client identity is provided by the client developer in the manifest file associated with the client;
- `S_LOGIN_CLIENT_IS_SERVICE`: the client is another Secure World service. The client identity assigned to this session is the calling service's UUID.

A service can retrieve the `sm.client.login` property in order to perform access control. For example, a service can refuse to open a session for a client that is not identified.

B2.2.4 PROCESSES AND THREADS

A process is created for each service instance. If the service is multi-instance, the process is created for each new session. If the service is mono-instance, the process may be created at boot-time or when the first session is opened. This depends on the property `config.s.lazy_instantiation` defined by the service.

A process can run multiple threads. The scheduling of threads within a given process is **cooperative**; the processor control cannot be passed preemptively from one thread to another thread within a given process. Instead, the points in time when the context switches may occur are clearly identified in this specification:

- When a thread calls a "**preemptable**" function, the control may be transferred to another thread eligible for execution in the process;
- When a thread calls a "**non-preemptable**" function, the API guarantees that the control cannot be passed to another thread of the process.

This style of threads is sometimes known as "*fibers*". The advantage is that the programmer has complete control over scheduling; with as little non-deterministic behavior as possible, it is much easier to produce robust thread-safe code. This is important for security, as there will be less scope for bugs due to multi-threading.

The style of scheduling between processes is implementation-defined. Some implementations schedule processes in a cooperative way. Others can preempt a process at any time and give the control to another process. This has little impact on the semantics of the API and the code of services, except that in a system with cooperative inter-process scheduling, each service should regularly call the function `SThreadYield` in order to give other processes a chance to run.

Each process has a dedicated memory heap, and memory heaps belonging to different processes are logically separated. Whether this separation is enforced is implementation dependent, but processes should never try to use pointers or references belonging to another process.

When a process is destroyed, its memory heap is also destroyed.

A service instance can access its heap using the memory allocation functions in section B2.3.8, "*Memory Management* API". Helper functions are also provided to copy, fill, or compare memory blocks.

B2.2.5 SECURE SERVICE PROVIDER INTERFACE

Each service must provide the implementation of a few functions, collectively called the “Secure Service Provider Interface” and referred to as the “SRVX” interface. These functions are the entry points that are called to create the instance, notify the instance that a new client is connecting, notify the instance when the client invokes a command, etc.

Each time one of the SRVX functions is called, it is called in a new distinct thread created by the system. This thread can create and control new threads using the Thread API (see section B2.3.10, “*Thread API*”). When the SRVX function returns, the thread created to run it is destroyed by the System. If the SRVX function started other threads, they are left running.

Here is the list of all the functions in the SRVX Interface:

- `SRVXCreate`: this is the service constructor. It is called once and only once in the life-time of the service instance. If this function fails, the instance is not created.
- `SRVXOpenClientSession`: this function is called whenever a new client attempts to connect to the instance. If this function fails, the connection is rejected. The service may optionally create a session context, which is an opaque memory block containing any session local data. The session context is associated with the connection, and a reference to it is passed by the implementation to all subsequent SRVX calls within the session that created the context. If a service instance supports multiple sessions, then this entry point is called in a new thread for each new connection. This means there can be multiple threads running this entry point simultaneously, subject to the restrictions for cooperative scheduling. If the service does not support multiple sessions, the system will reject all new connection while a session is already active.
- `SRVXInvokeCommand`: this function is called when a client invokes a service command. The session context reference is given back to the service by the system. If the service supports multiple commands, then this entry point is called in a new thread for each new command. This means that there can be multiple threads running this entry point. If the service does not support multiple commands, the system will queue any new command while a command is already active.
- `SRVXCloseClientSession`: this function is called when the client disconnects. The implementation guarantees that there are no active commands in the session being closed. The session context reference is given back to the service by the system. It is the responsibility of the service to deallocate the session context if memory has been allocated for it.
- `SRVXDestroy`: this is the service destructor. This function is called just before the instance is terminated. It is guaranteed that no client sessions are open when this function is called by the system. After `SRVXDestroy` returns the system collects all the resources opened by the process. All threads are destroyed, all files, keys, and cryptographic sessions are closed, and all of the heap memory is reclaimed.

The following table summarizes the correspondence between the functions that the client calls and the functions of the service that the system calls:

Table B2-1: SSDI: Correspondance Client Functions ↔ SRVX Functions

Operation in the client:	Effects on the service:

`TEEC_OpenSession`
or
`SXControlOpenClientSession`
	`SRVXCreate` is called (only if a new instance is created to handle the session)
	`SRVXOpenClientSession` is called

`TEEC_InvokeCommand`
or
`TEEC_InvokeCommandEx`
or
`SXControlInvokeCommand`
	`SRVXInvokeCommand` is called

`TEEC_CloseSession` *or*	`SRVXCloseClientSession` is called `SRVXDestroy` is called (only if the instance

`SHandleClose` called on a session handle	is destroyed as a result of the session being closed)
`TEEC_RequestCancellation` *or* operation times out *or* secure calling thread is cancelled	The thread currently calling `SRVXOpenClientSession` or `SRVXnvokeCommand` is cancelled.

When a client opens a session or invokes a command, it can attempt to cancel the operation after it has started. The system reacts by setting the cancellation state of the corresponding service thread, i.e., the thread that is currently executing the SRVX function of the cancelled operation. The service thread can poll its cancellation state. It can also configure itself so that some API functions are interrupted when the thread is cancelled. The API functions that may react to cancellation are called "**cancellable**".

It is not mandatory for a service to implement a cancel procedure. It may well just ignore the cancellation requests, and return once the operation has completed as normal. This is the default behavior.

When a client dies unexpectedly the system detects the death and allows the service to act as if the client had disconnected cleanly; all the pending commands are cancelled, the system waits until they are completed, and then finally closes the client session.

B2.2.6 Operation Parameters in the SRVX Interface

When a Client opens a session on a service or invokes a command, it can send *Operation Parameters* to the service. blThe parameters encode the data associated with the operation.

Up to four parameters can be sent in an operation. Each parameter has a type, which encodes the nature of the parameter and its direction. It can be one of the following types:

- The type `NONE` is used to denote that the parameter is not used.
- The types `VALUE_INPUT`, `VALUE_OUTPUT`, or `VALUE_INOUT` denote "Value Parameter" containing two 32-bit integers named `a` and `b`. The type also encodes the direction of data flow:
  - An input value transmits the integers only from the client to the service.
  - An output value allows a service to transmit the integers to its client.
  - An inout value allows both communications.
- The types `MEMREF_INPUT`, `MEMREF_OUTPUT`, or `MEMREF_INOUT` denote "Memory Reference Parameters" containing a pointer to a buffer and its associated size. In some implementations the buffer is directly shared with the client. The parameter type encodes the direction of data flow:
  - An Input Memory Reference only allows the service to read in the buffer. The buffer can be `NULL`
  - An Output Memory Reference allows the service to write into the buffer and also to update the size of the buffer:
    - If the updated size is smaller or equal to the original size, it encodes the actual number of bytes written in the buffer
    - If the updated size is greater than the original size, it is a means for the service to indicate that the buffer is too small and to transmit the desired buffer size. In this case, it is not guaranteed that whatever has been written in the buffer will be visible to the client.

      Note that an Output Memory Reference can be `NULL`, in which case the parameter is typically used to request the necessary size of the output buffer and the operation will be followed by a new operation with a non-`NULL` parameter.
  - An Inout Memory Reference combines the characteristics of both an Input and an Output Memory Reference. It can be used by the client to transmit input data to the service and by the service to transmit output data to the client or to indicate the necessary output buffer size.

Note that Memory Reference Parameters typically point to memory owned by the client and shared with the service for the duration of the operation. This is useful to minimize the number of memory copies and the data footprint in case a service must deal with large data buffers, for example to process a multimedia stream protected by DRM.

The fact that Memory References may use memory directly shared with the client implies that the service must be especially careful when handling such data: even if the client is not allowed to access the shared memory buffer during an operation on this buffer, the system usually cannot enforce this restriction. A badly-designed or rogue client may well change the content of the shared memory buffer at any time, even between two consecutive memory accesses by the service. This means that the service should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the service should always read data only once from a shared buffer and then validate it. It must not assume that data written to the buffer can be read unchanged later on.

B2.2.7 SESSION THREADS

Some functions of the API can only be called from a session thread. A session thread is a thread created by the system when it calls one of the following entry points:

- `SRVXOpenClientSession`,
- `SRVXInvokeCommand`,
- `SRVXCloseClientSession`.

A few functions of the API implicitly depend on the identity and properties of the “current client”. These functions can only be called from a session, because only session threads are implicitly bound to a client session.

Here is the list of all the functions that can only be called from a session thread:

- All the `SSessionXXX` functions
- `CV_ActivateUpdateShortcut`

If the service calls one of these functions from a non-session thread, it will panic and the instance destroyed. However, if such a function returns a handle on an object, this handle can then be used from any thread within the process, even threads associated with another client.

B2.2.8 ERROR HANDLING AND PANICS

The API functions usually return an error code of type `S_RESULT` to indicate errors to the caller.

In case of critical errors the functions do not return to the caller but “**panic**”. In general, the API functions panic when a programmatic error is detected (wrong parameters, wrong state, etc.)

When an API function panics, the system kills the process; it terminates all the pending commands, disconnects all the client sessions, closes all the resources that the service instance opened, and destroys the instance. Note that multiple processes can reference a common resource, for example a file. If a processes sharing a resource terminates the system does not destroy the resource immediately, but will wait until no other processes reference the resource before reclaiming it.

After a panic, no SRVX function of the instance is ever called again, not even `SRVXDestroy`.

From the client’s point of view, when a service panics, the Client API returns the TEE error `TEEC_ERROR_TARGET_DEAD`. If the client runs in the Secure World and accesses the service using the `SXControl` API, then the API returns the error code `S_ERROR_TARGET_DEAD`.

The SRVX functions, implemented by the service, return an error code of type `S_RESULT`. The behavior of the system when an entry point returns an error depends on what entry point is called:

- If `SRVXCreate` returns any error, the instance is not created;
- `SRVXDestroy` returns no error;
- If `SRVXOpenClientSession` returns any error code, the client connection is rejected. Additionally, the error code is returned to the client with the origin set to `TEEC_ORIGIN_TRUSTED_APP`;
- If `SRVXInvokeCommand` returns an error code, this error code is returned to the client with the origin set to `TEEC_ORIGIN_TRUSTED_APP`;
- `SRVXCloseClientSession` returns no error.

This procedure also applies to Secure World clients.

B2.2.9 THREAD API

Threads are created by the system when a service entry point is called, and they can be explicitly created by the service itself.

The Thread API contains functions to create and manage threads. A service can create threads, wait for thread completion, cancel threads, yield control from the current thread or sleep.

B2.2.9.1 THREAD STACK USAGE

Whenever a thread is created by the system, or by a service, it is allocated a fixed size stack within the instance heap.

Threads created by the system to run the SRVX entry points are created with the stack size controlled in the service property `config.s.default_stack_size`. This property describes the actual stack size available to the user code, excluding any stack size necessary for the system. For example, if the system requires 512 bytes of stack for all the API functions as well as for the code calling the thread's entry point and if the programmer sets the `config.s.default_stack_size` property to 128 bytes, then the system will internally allocate a stack of 640 bytes to the thread. The user code will therefore have 128 bytes fully available.

Threads created by the service itself have a stack size that is either `config.s.default_stack_size` bytes or a size specified when the thread is created.

The implementation does not enforce the limit of the stack, and as such stack overflow is likely to cause corruption of memory in the process heap. In systems without enforced inter-process separation, it may also cause corruption of other processes memory. It is therefore critical that stack limits are never exceeded by a service.

B2.2.9.2 THREAD CANCELLATION

A service can cancel one of the threads it has created using the function `SThreadCancel`. The system can also cancel a thread running some of the service's entry-points.

As stated previously, it is not mandatory for a service to implement a cancel procedure. It may well just ignore the cancellation requests and continue as normal. However, the Thread API provides function for managing thread cancellation when required.

Each thread has an implicit cancellation state that can be viewed as a boolean indicating whether the thread has been cancelled. A thread can use the function `SThreadIsCancelled` to check whether it has been cancelled or not. A thread can then decide to react to a cancellation when the cancellation state is set.

Any thread can configure itself so that cancellable functions of the API are interrupted when the thread is cancelled. By default the cancellation state of a new thread is masked. This means that when a thread has been marked as cancelled and it calls a cancellable function, the execution of the function will not be interrupted and will proceed normally. If cancellation requests are not masked and if a thread is marked as cancelled, then when the thread calls a cancellable function, the function will return the error code `S_ERROR_CANCEL`. A thread can mask and unmask cancellation requests by calling the function `SThreadMaskCancellation`.

A thread can also reset its cancellation state by calling the function `SThreadResetCancel`. After this function has been called, the thread will no longer be cancelled.

Here is the list of all cancellable functions of this API:

- `SThreadSleep`
- `SThreadJoin`
- `SSemaphoreAcquire`
- `SXControlOpenClientSession`
- `SXControlInvokeCommand`

All cancellable functions can also take a timeout, after which the system will attempt to cancel the function.

The timeout value is represented by an absolute time limit. The use of an absolute limit is useful when trying to apply a timeout to a sequence of several functions. This absolute time limit is manipulated through the `S_TIME_LIMIT` structure, which is an opaque structure. The service can call the function `STimeGetLimit` to convert a relative timeout value to an absolute time limit encoded in the structure `S_TIME_LIMIT`.

The time limit is set with reference to the Secure World system time. In many implementations the Secure World may not have access to real secure time hardware, such as a timer or a real time clock, and is forced to derive its time from the Normal World operating system. In these cases the system time is only as trustworthy as the Normal World operating system, and as such the timeouts should not be relied on for security critical behavior.

B2.2.9.3 THREAD SYNCHRONIZATION

This API defines semaphores for synchronizing threads. Semaphore objects are decrementing and counting. When a semaphore object is created, its counter is set to an initial value provided by the service. The function `SSemaphoreAcquire` is used by a thread to work out if it is allowed in to a critical section. If the count is non-zero this function decrements the semaphore by one, and allows the calling thread to proceed into the critical section. If the semaphore count is 0, the calling thread is suspended and will wait until it can acquire the semaphore before continuing. The function `SSemaphoreRealease` increments the semaphore count by one, allowing another process to enter the critical section if the count was previously zero.

B2.2.9.4 SAMPLE USAGE

The following is a sample code that demonstrates thread basics. This sample shows a command that creates a thread, does something, cancel the thread and wait for the thread termination before returning. The thread created in this example is very simple and shows how a thread can handle cancellation requests from other threads. This thread does some action in loop until it receives a cancellation request.

```
uint32_t my_thread(void* pArgs)
{
   S_RESULT nError;
   uint32_t nCode;
   S_HANDLE hSem;

   hSem = (S_HANDLE)pArgs;

   /* Unmask cancellation requests */
   SThreadMaskCancellation(false);

   while(true)
   {
      S_TIME_LIMIT sTimeLimit;

      /* Check if the thread is cancelled */
      if (SThreadIsCancelled(&nCode))
      {
         return 0;
      }

      /* Do Something */

      /* Signal the main thread of an event */
      SSemaphoreRelease(hSem);

      /*
       * Do Something. For example, call a cancellable function.
       * This will also yield the current thread.
       */
      STimeGetLimit(&sTimeLimit, 1000);
```

```
        nError = SThreadSleep(&sTimeLimit);
        if (nError == S_ERROR_CANCEL)
        {
            /* Thread has been cancelled */
            return 0;
        }

        /* Do Something */
    }
    return 0;
}


S_RESULT SRVX_EXPORT SRVXInvokeCommand(
    IN OUT void*    pSessionContext,
           uint32_t nCommandID,
           uint32_t nParamTypes,
           S_PARAM  pParams[4])
{
   S_HANDLE     hThread;
   S_HANDLE     hSem;
   S_RESULT     nError;
   /* Do something */

   /* Create the semaphore */
   nError = SSemaphoreCreate(0, &hSem);
   if (nError != S_SUCCESS)
   {
      /* Handle errors */
   }

   /* Create the thread */
   nError = SThreadCreate(&hThread, 0, my_thread, (void*)hSem);
   if (nError != S_SUCCESS)
   {
      /* Handle errors */
   }

   /* Do something */

   /* Wait for a signal of the other thread. The NULL time-limit
      denotes an infinite time-out */
   nError = SSemaphoreAcquire(hSem, NULL);

   /*
    * Since this thread has masked cancelation and
    * that timeout is infinite, there should not be error
    */
   SAssert(nError == S_SUCCESS);

   /* Cancel the thread */
   SThreadCancel(hThread, 0);

   /* Wait for the thread to complete */
   nError = SThreadJoin(hThread, NULL);
   if (nError != S_SUCCESS)
   {
      /* Handle errors */
   }

   /* Do something */
   return S_SUCCESS;
}
```

B2.2.10 File API

Each service has its own private file-system. This file-system is shared between all the instances of the service. The file-system is **flat**, which means that there is no directory structure within it.

The precise level of security or robustness of the file-system may depend on your product or its integration on a specific platform. However, the following properties should be expected from any implementation:

- **Confidentiality**: data stored in the file system should never be visible in clear-text from the Normal World;
- **Integrity** and **Authenticity**: while it is generally not possible to ensure that the data stored in the file-system is protected in integrity, any implementation should detect breaches of integrity and report them by using the error code `S_ERROR_STORAGE_CORRUPTED`. Whether this detection extends to replay attacks depends on your integration;
- **Device Binding**: data stored in the file-system on one device is not moveable to another device;
- **Access Control**: normal uses of the File API ensure that a service cannot access the file-system of another service. Whether this separation is enforced in all circumstances is implementation-dependent. However, all implementations guarantee that data stored by secure services cannot be accessed from the Normal World;
- **Atomicity**: modifications in the file-system are guaranteed to be atomic, which means that if power is lost while the operation is performed, the implementation ensures that either the whole operation is completed, or if this is not possible, that the partially complete operation is rolled back to the previous state.

Files are manipulated through file handles, which are created by use of the function `SFileOpen`. Each file handle contains a current file position, used to determine what byte in the file will be next read or written.

File names have a maximum size of 64 bytes excluding the zero-terminating character.

B2.2.11 Cryptographic API

Each service is associated with two private key stores that can be used through the Internal Cryptographic API, a subset of PKCS#11 defined in Chapter B4 "*Trusted Foundations Cryptographic API*". These keystores are called the "system private service keystore" and the "user private service keystore" respectively.

These keystores are seen by the service as virtual tokens in this API. They are logically separated, i.e., objects stored in the system keystore cannot be accessed in the user keystore. Depending on the implementation, the user keystore may use a physical storage that is separate from the system keystore and may have a different life-cycle. See your *Product Reference Manual* of your product for more information.

Note that the persistency of the key stores of the Crypto API is built around the same foundations as the File API, and therefore provides the same security and robustness guarantees.

The "Cryptoki Update Shortcut" allows a service to streamline the processing of multi-stage cryptographic operations by installing an "update shortcut". Once installed, such a shortcut directly handles some commands issued by clients of the service and automatically calls the Cryptographic API on behalf of the service. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

B2.2.12 Date API

Each service is associated with a private persistent date reference. This allows a service to synchronize its own date separately from the other services.

Initially the date reference of a service is not set and the date returned by the date API must be interpreted as an absolute time. Once the service has set its date reference, the date API returns the current date of the service. Note that the time origin for a given date reference is arbitrarily chosen by the service. For example, when a service sets its current date to 1000 seconds through the function `SDateSet`, it is the service responsibility to determine what the origin of this date is (for example midnight of January 1, 1970). Typically the time origin is chosen in accordance with the time origin used by a given time server.

The function `SDateGet` also returns the current status of the date that may indicate a corruption of the date source. The meaning of corruption depends on the protection level provided by the date API.

Services can retrieve the protection level of the implementation through the implementation property `date.protectionLevel`. When a date source appears to be corrupted, it is recommended that the service resynchronize the date with a trusted time source.

When a service cannot synchronize with an external time server, it can use the function `SClockGet` to retrieve an absolute wall-clock time from the local device. The value returned by this function should not be trusted because it might be derived from non-trusted components, including the Normal World operating system's time. Moreover, it may not benefit from the same protection level that is provided by the date API. It should only be used when synchronization with a trusted-time source is not possible. The following code sample shows how a service can initialize its date reference with the wall-clock time value:

```
/* Retrieve the current date */
nError = SDateGet(&nSeconds, &nStatus, 0);
if (nError == S_SUCCESS)
{
   if (nStatus == S_DATE_STATUS_NOT_SET)
   {
      /*
       * If the date has not been set, set the date reference
       * with the current wall-clock time.
       */
      nError = SDateSet(SClockGet(), 0);
      if (nError != S_SUCCESS)
      {
         /* Handle errors */
      }
   }
   else if (nStatus == S_DATE_STATUS_NEEDS_RESET)
   {
      /* Handle date corruption here */
      /* Reset the date */
      nError = SDateSet(SClockGet(), 0);
      if (nError != S_SUCCESS)
      {
         /* Handle errors */
      }
   }
   else /* S_DATE_STATUS_SET */
   {
       /* Date successfully retrieved, do something . . . */
   }
}
```

B2.2.13 Service Control API

The Service Control API allows a service to act as a client of another service. It is similar to the TF Client API (see Chapter B3, "*Trusted Foundations Client API*"), but available to the services instead of the external applications.

Note that the Service Control API can be used to access the Service Manager, using the protocol specified in Chapter B7, "*Service Manager Protocol*".

B2.2.14 Power Management

Service developers must be aware that the system may go into a low-power mode at any time. This means that some secure memory, some peripherals, or even the core itself can be shut down to save battery. In this case, it is always guaranteed that the system takes the necessary steps to save the state of the Secure World in a secure way and to restore this state when an operation must be resumed. This is transparent from a service point of view.

When the system shuts down completely, the behavior of the system is implementation-dependent. Refer to your *Product Reference Manual* for more details.

B2.3 SSDI REFERENCE

This section provides a complete reference of the Secure Service Development Interface.

B2.3.1 TYPES

See section B1.1, “*Common Types*” for the definition of basic types

B2.3.1.1 S_PROPERTY

```
typedef struct
{
   char* pName;
   char* pValue;
} S_PROPERTY;
```

This structure contains a property, which is a name-value pair. The name and value are Unicode strings encoded as zero-terminated UTF-8.

B2.3.1.2 S_CALENDAR

```
typedef struct
{
   int32_t  nYear;
   int32_t  nMonth;
   int32_t  nDayOfWeek;
   int32_t  nDay;
   int32_t  nHour;
   int32_t  nMinute;
   int32_t  nSecond;
} S_CALENDAR;
```

This structure represents a date and time with individual members for the various elements of the date.

A calendar structure is said to be normalized when its fields satisfy the following range constraints:

- `nMonth`: from 1 (January) to 12 (December)
- `nDayOfWeek`: from 0 (Sunday) to 6 (Saturday)
- `nDay`: from 1 to 31
- `nHour`: from 0 to 23
- `nMinute`: from 0 to 59
- `nSeconds`: from 0 to 59

A non-normalized calendar structure is sometimes useful for date computation. See in particular the functions `SDateConvertSecondsToCalendar` and `SDateConvertCalendarToSeconds`.

B2.3.1.3 S_FILE_INFO

```
typedef struct
{
   char*    pName;
   uint32_t nSize;
   uint32_t nNameLength;
} S_FILE_INFO;
```

This structure contains information about a file:

- `pName` is the name of the file encoded in a zero-terminated string
- `nSize` is the size in bytes of the file
- `nNameLength` is the number of bytes in the file name, excluding the terminating zero

B2.3.1.4 S_WHENCE

```
typedef enum
{
   S_FILE_SEEK_SET = 0,
   S_FILE_SEEK_CUR,
   S_FILE_SEEK_END
} S_WHENCE;
```

Enumerates the possible start offset when moving a file pointer.

B2.3.1.5 S_TIME_LIMIT

```
typedef struct
{
   uint32_t nTime1;
   uint32_t nTime2;
} S_TIME_LIMIT;
```

The time limit type denotes an absolute time and date. It is used by the functions that are aborted after a timeout limit

All instances of this structure must be created through the function `STimeGetLimit`; otherwise the implementation behavior is undefined. The content of the two fields, `nTime1` and `nTime2`, is implementation-specific and must not be interpreted or modified by the caller.

B2.3.1.6 S_PARAM

```
typedef union
{
   struct
   {
      void*    pBuffer;
      uint32_t nBufferSize;
   } memref;
   struct
   {
      uint32_t a, b;
   } value;
} S_PARAM;
```

This union describes one parameter passed by the system to the entry points `SRVXOpenClientSession` or `SRVXInvokeCommand` or passed by a service to the Service Control API functions `SXControlOpenClientSession` or `SXControlInvokeCommand`.

Which of the field `value` or `memref` to select is determined by the parameter type specified in the argument `nParamTypes`. See the section B2.3.4.6, "*Operation Parameters in the SRVX Interface*" and the section B2.3.15.3, "*Operation Parameters in the SXControl API*" for more details on how this type is used.

B2.3.2 FORMAT OF PROPERTIES

This specification defines implementation properties, service properties and session properties.

A property is a name-value pair. Each property value is a zero-terminated string of Unicode characters encoded in UTF-8. However, some properties are interpreted as having particular types, like integers or Booleans. This section provides the definition of how strings are interpreted as typed items when necessary.

B2.3.2.1 INTEGERS

Integers can be expressed in decimal, hexadecimal, or binary form:

```
integer:                decimal-integer
                      | hexadecimal-integer
                      | binary-integer

decimal-integer:        +[0-9]
hexadecimal-integer:    0[x,X]+[0-9,a-f,A-F]
binary-integer:         0[b,B]+{0,1}
```

Integers are always unsigned and must be representable on 32 bits, so they can range from 0 to 2^{32}-1.

B2.3.2.2 Booleans

Boolean properties must have the value “`true`” or “`false`”, case-insensitive.

B2.3.3 Constants

B2.3.3.1 Error Codes

See section B1.2, “*Common Error codes*”.

B2.3.3.2 Parameter Types

Table B2-2: SSDI Parameter Type Constants

S_PARAM_TYPE_NONE	0
S_PARAM_TYPE_VALUE_INPUT	1
S_PARAM_TYPE_VALUE_OUTPUT	2
S_PARAM_TYPE_VALUE_INOUT	3
S_PARAM_TYPE_MEMREF_INPUT	5
S_PARAM_TYPE_MEMREF_OUTPUT	6
S_PARAM_TYPE_MEMREF_INOUT	7

B2.3.3.3 Login Types

Table B2-3: SSDI Login Type Constants

S_LOGIN_PUBLIC	0x00000000
S_LOGIN_USER	0x00000001
S_LOGIN_GROUP	0x00000002
S_LOGIN_APPLICATION	0x00000004
S_LOGIN_APPLICATION_USER	0x00000005
S_LOGIN_APPLICATION_GROUP	0x00000006
S_LOGIN_AUTHENTICATION	0x80000000
S_LOGIN_PRIVILEGED	0x80000002
S_LOGIN_CLIENT_IS_SERVICE	0xF0000000

B2.3.3.4 Other Constants

Table B2-4: SSDI Other Constants

Constant Name	Value
S_ORIGIN_API	0x00000001
S_ORIGIN_COMMS	0x00000002

S_ORIGIN_TEE	0x00000003
S_ORIGIN_TRUSTED_APP	0x00000004
S_CRYPTOKI_KEYSTORE_PRIVATE	0x00000001
S_CRYPTOKI_KEYSTORE_PRIVATE_USER	0x00004004
S_DATE_STATUS_NOT_SET	0xFFFF5000
S_DATE_STATUS_NEEDS_RESET	0xFFFF5001
S_DATE_STATUS_SET	0x00000000
S_FILE_STORAGE_PRIVATE	0x00000001
S_FILE_FLAG_ACCESS_READ	0x00000001
S_FILE_FLAG_ACCESS_WRITE	0x00000002
S_FILE_FLAG_ACCESS_WRITE_META	0x00000004
S_FILE_FLAG_SHARE_READ	0x00000010
S_FILE_FLAG_SHARE_WRITE	0x00000020
S_FILE_FLAG_CREATE	0x00000200
S_FILE_FLAG_EXCLUSIVE	0x00000400
S_FILE_MAX_POSITION	0xFFFFFFFF
S_FILE_NAME_MAX	64
S_TIMEOUT_INFINITE	0xFFFFFFFF
S_UPDATE_SHORTCUT_FLAG_AGGRESSIVE	0x00000001

B2.3.4 SERVICE INTERFACE

Every service must implement and export the entry point functions defined in this sub-section of the specification, which are known as the Service Interface functions or the SRVX entry points and are prefixed with the keyword `SRVX`.

The system will call these entry points whenever a client triggers some behavior which requires action from the service. For example, when a client wants to connect to a multi-instance service, the system will first call `SRVXCreate` to construct a new instance, followed by a call to `SRVXOpenClientSession` to create the new client session.

All of the SRVX functions are always called by the system in a newly created thread.

The implementation of these service entry points must use the keyword `SRVX_EXPORT`, as shown in the function prototypes in this section. This keyword will expand to a compiler-specific C keyword which will export the symbol appropriately.

B2.3.4.1 SRVXCREATE

```
S_RESULT SRVX_EXPORT SRVXCreate( void )
```

The function `SRVXCreate` is the service constructor, which the system calls when it creates a new instance of the service.

The implementation can use the function `SInstanceSetData` to register instance data.

This function is called:

- At boot-time for mono-instance services.
- When a first client session is opened for mono-instance services with lazy instantiation.
- Each time a client session is opened for multi-instance services.

In any case, the system guarantees that the thread calling `SRVXCreate` is never cancelled, even if the instance is created as a result of the client opening a session and this operation is cancelled.

Return Value

- `S_SUCCESS`: if the instance is successfully created, the function must return `S_SUCCESS`.
- Any other value: if any other code is returned the instance is not created, and no other function entry points of this instance will be called. The system will reclaim the heap of the process and will dereference all objects referenced through the process.

  If the instance was created as a result of a client's opening a session, the error code is returned to the client.

B2.3.4.2 SRVXDESTROY

```
void SRVX_EXPORT SRVXDestroy( void )
```

The function `SRVXDestroy` is the service destructor, which the system calls when the instance is being destroyed.

This function is called:

- When the system is shut down for a mono-instance service.
- When the last session is closed for a mono-instance service with lazy instantiation.
- When the session is closed for a multi-instance service.

When the function `SRVXDestroy` is called, the system guarantees that no client session is currently open. Once the call to `SRVXDestroy` has been completed, no other entry point of this instance will ever be called.

The implementation can use the function `SInstanceGetData` to access instance data.

The system guarantees that the thread that calls `SRVXDestroy` is never cancelled.

Return Value

This function can return no success or error code. When this function returns the system will consider that the instance is destroyed and will reclaim all resources left open by the instance.

B2.3.4.3 SRVXOpenClientSession

```
S_RESULT SRVX_EXPORT SRVXOpenClientSession(
              uint32_t nParamTypes,
       IN OUT S_PARAM pParams[4],
       OUT    void**   ppSessionContext )
```

The system calls the function `SRVXOpenClientSession` when a new client connects to the service instance.

The client can specify parameters in an open operation and these parameters are passed to the service in the arguments `nParamTypes` and `pParams`. These arguments can also be used by the service to transfer answer data back to the client. See section B2.3.4.6 for a specification of how to handle the operation parameters.

If this function returns `S_SUCCESS`, the client is connected and can invoke service commands. When the client disconnects, the system will eventually call the `SRVXCloseClientSession` entry point.

If the function returns any error, the system rejects the connection and returns the error code and the current content of the parameters the client. The return origin is then set to `TEEC_ORIGIN_TRUSTED_APP` (or `S_ORIGIN_TRUSTED_APP` if the client is a secure service).

The service can register a session data pointer by setting `*ppSessionContext`. The value of this pointer is not interpreted by the system, and is simply passed back to other SRVX functions within this session. Note that `*ppSessionContext` may be set with a pointer to a memory allocated by the service or with anything else, like an integer, a handle etc. The system will *not* automatically free `*ppSessionContext` when the session is closed; the service is responsible for freeing memory if required.

During the call to this function the client may cancel the operation. In this case, the system cancels the thread running the function as if it called the function `SThreadCancel`. The service may still ignore the cancellation request and return `S_SUCCESS`. The client must then consider the session as successfully created and explicitly close it if necessary.

Parameters

- `nParamTypes`: the types of the four parameters. See section B2.3.4.6 for more information
- `pParam`: a pointer to an array of four parameters. See section B2.3.4.6 for more information
- `ppSessionContext`: A pointer to a variable that can be filled by the service with an opaque `void*` data pointer

Return Value

- `S_SUCCESS` if the session is successfully created.
- Any other value if the session could not be open.
    - The error code may be one of the pre-defined codes (see section B2.3.3.1, "*Error Codes*"), or may be a new error code defined by the service implementation itself.

B2.3.4.4 SRVXCloseClientSession

```
void SRVX_EXPORT SRVXCloseClientSession(
    IN OUT void*    pSessionContext)
```

The system calls this function to indicate that a session is being closed. During the call to this function the implementation can use any session functions.

When it calls this function, the system guarantees that there are no outstanding commands being invoked by the session. If the client disconnects while invoking commands, the system cancels the commands and waits for this to complete before it calls `SRVXCloseClientSession`.

Closing a session is not cancellable. The system guarantees that the thread running `SRVXCloseClientSession` will never be cancelled.

The service implementation is responsible for freeing any resources consumed by the session being closed. Note that the service cannot refuse to close a session, but can hold the the closing until it returns from `SRVXCloseClientSession`. This is why this function cannot return an error code.

When this function returns, all remaining cryptoki update shortcuts associated with this client session are deactivated. See section B2.3.14.1, “*Cryptoki Update Shortcut*” for more information.

Parameters

- `pSessionContext`: The value of the `void*` opaque data pointer set by the service in the function `SRVXOpenClientSession` for this session.

B2.3.4.5 SRVXInvokeCommand

```
S_RESULT SRVX_EXPORT SRVXInvokeCommand(
    IN OUT void*     pSessionContext,
           uint32_t nCommandID,
           uint32_t nParamTypes,
    IN OUT S_PARAM  pParams[4] )
```

The system calls this function when the client invokes a command within a session opened on the instance.

The service can access the parameters sent by the client through the `nParamTypes` and `pParams` arguments. It can also use these arguments to transfer answer data back to the client. See section B2.3.4.6 for a specification of how to handle the operation parameters.

During the call to this function the client may cancel the command. In this case the system cancels the thread running the function as if it had called the function `SThreadCancel`.

A command is always invoked within the context of a client session. In the thread that runs the `SRVXInvokeCommand` entry point, the implementation can call any session functions.

Parameter

- `pSessionContext`: The value of the `void*` opaque data pointer set by the service in the function `SRVXOpenClientSession`.
- `nCommandID`: A service-specific code that identifies the command to be invoked.
- `nParamTypes`: the types of the four parameters. See section B2.3.4.6 for more information
- `pParam`: a pointer to an array of four parameters. See section B2.3.4.6 for more information

Return Value

- `S_SUCCESS`: if the command is invoked successfully, the function must return this value.
- Any other value: if the invocation of the command fails for any reason.
  - The error code may be one of the pre-defined codes (see section B2.3.3.1), or may be a new error code defined by the service implementation itself.

B2.3.4.6 Operation Parameters in the SRVX Interface

When a client opens a session or invoke a command within a session, it can transmit operation parameters to the service and receive answer data back from the service. The system uses the arguments `nParamTypes` and `pParams` to encode the parameters and their types and pass them to the service. While executing the entry point, the service can also write in `pParams` to encode the answer data.

Parameter Types

The argument `nParamTypes` encodes the type of each of the four parameters passed to an entry point. The content of `nParamTypes` is described in Table B2-5:

Table B2-5: SSDI Content of nParamTypes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU, set to 0																Type[3]				Type[2]				Type[1]				Type[0]			

Each parameter type can take one of the `S_PARAM_TYPE_XXX` values listed in Table B2-2. The type of each parameter determines whether the parameter is used or not, whether it is a Value or a Memory Reference, and the direction of data flow between the Client and the Service: Input (Client to Service), Output (Service to Client), or both Input and Output.

The following macros are available in `ssdi.h` to decode `nParamTypes`:

```
#define S_PARAM_TYPES(t0,t1,t2,t3) \
   ((t0) | ((t1) << 4) | ((t2) << 8) | ((t3) << 12))

#define S_PARAM_TYPE_GET(t, i) (((t) >> (i*4)) & 0xF)
```

The macro `S_PARAM_TYPES` can be used to construct a value that you can compare against an incoming `nParamTypes` to check the type of all the parameters in one comparison, like in the following example:

```
if (nParamTypes !=
            S_PARAM_TYPES(
                  S_PARAM_TYPE_MEMREF_INPUT,
                  S_PARAM_TYPE_MEMREF_OUPUT,
                  S_PARAM_TYPE_NONE,
                  S_PARAM_TYPE_NONE))
{
   /* Bad parameter types */
   return S_ERROR_BAD_PARAMETERS;
}
```

The macro `S_PARAM_TYPE_GET` can be used to extract the type of the *i*th parameter from `nParamTypes` if you need more fine-grained type checking.

Initial Content of pParams

When the system calls the service entry point, it initializes the content of `pParams[i]` as described in Table B2-6.

Table B2-6: SSDI Content of pParams[i] when calling a service entry point

Value of Type[i]	Content of pParams[i] when the entry point is called
`S_PARAM_TYPE_NONE` `S_PARAM_TYPE_VALUE_OUTPUT`	Filled with zeros
`S_PARAM_TYPE_VALUE_INPUT` `S_PARAM_TYPE_VALUE_INOUT`	`pParams[i].value.a` and `pParams[i].value.b` contain the two integers sent by the client

`S_PARAM_TYPE_MEMREF_INPUT`
`S_PARAM_TYPE_MEMREF_OUTPUT`
`S_PARAM_TYPE_MEMREF_INOUT`
	`pParams[i].memref.pBuffer` is a pointer to memory buffer shared by the client. This can be `NULL`.
	`pParams[i].memref.nSize` describes the size of the buffer. If `pBuffer` is `NULL`, `nSize` is guaranteed to be zero.

Note that if the client uses the Normal World TF Client API, the service cannot distinguish between a registered and a temporary memory reference. Both are encoded as one of the `S_PARAM_TYPE_MEMREF_XXX` types and a pointer on the data is passed to the service.

Security Warning: *For a Memory Reference parameter, the buffer may concurrently exist within the client memory space. It must therefore be assumed that the client is able to change to content of this buffer asynchronously at any moment. It is a security risk to assume otherwise.*

Any service which implements functionality that needs some guarantee that the contents of a buffer are constant should copy the contents of a shared buffer into service-owned memory.

Behavior of the System when the Service Returns

When the Service entry point returns, the system reads the content of each `pParams[i]` to determine what answer data to send to the client.

Value of Type[i]	Behavior of the system when entry point returns

`S_PARAM_TYPE_NONE`
`S_PARAM_TYPE_VALUE_INPUT`
`S_PARAM_TYPE_MEMREF_INPUT`
	The content of `pParams[i]` is ignored

`S_PARAM_TYPE_VALUE_OUTPUT` `S_PARAM_TYPE_VALUE_INOUT`	`pParams[i].value.a` and `pParams[i].value.b` contain the two integers sent to the client
`S_PARAM_TYPE_MEMREF_OUTPUT` `S_PARAM_TYPE_MEMREF_INOUT`	The system reads `pParams[i].memref.nSize`: • If it is equal or less than the original value of `nSize`, it is considered as the actual size of the memory buffer. In this case, the system assumes that the service has not written beyond this actual size and only this actual size will be synchronized with the client • If it is greater than the original value of `nSize`, it is considered as a request for a larger buffer. In this case, the system assumes that the service has not written anything in the buffer and no data will be synchronized.

Memory Reference Parameters and Memory Synchronization

If a parameter is a Memory Reference, you must be aware that the memory buffer may be released or unmapped immediately after the operation completes. Also, some implementations may explicitly synchronize the contents of the memory buffer before the operation starts and after the operation completes.

As a consequence:

- the service must not access the memory buffer after the operation completes. In particular, it cannot be used as a long-term communication means between the client and the service. You must only access a memory reference during the lifetime of the operation;
- the service must not attempt to write into a memory buffer of type `S_PARAM_TYPE_MEMREF_INPUT`;
- for a memory reference buffer marked as `OUTPUT` or `INOUT`, the service can write in the entire range described by the initial content of `pParams[i].memref.nSize`. However, the system only guarantees that the client will observe the modifications below the final value of `nSize` and only if the final value is equal or less than the original value.

  For example, assume the original value of `nSize` is 100:

  - If the service does not modify this value, the complete buffer is synchronized and the client is guaranteed to observe all the changes;
  - If the service writes 50 in `nSize`, then the client is only guaranteed to observe the changes within the range from index 0 to index 49;
  - If the service writes 200 in `nSize`, then no data is guaranteed to be synchronized with the client. However, the client will receive the new value of `nSize`. The service can typically use this feature to tell the client that the memory reference was too small and request that the client retry with a memory reference of at least 200 bytes.

Failure to comply with these constraints will result in undefined behavior.

B2.3.5 INSTANCE FUNCTIONS

As described in section B2.2.1, "*Developing Secure Services*", writable global variables are not supported by the system.

The functions in this section provide a way to manage global data. They allow setting and reading an instance variable which is accessible from any thread within the instance. Typically this instance variable will be used to hold a pointer to a service defined memory block, which contains any writable data that needs global scope, or writable static data that needs function scope.

In addition to this, an equivalent session context variable for managing session global and static exists for sessions (see `SRVXOpenClientSession`, `SRVXInvokeCommand`, and `SRVXCloseClientSession`).

B2.3.5.1 SINSTANCESETDATA

```
void SInstanceSetData( IN void* pInstanceData )
```

This function sets the instance data pointer. This can be retrieved later by any thread of the instance by calling the `SInstanceGetData` function.

This function can be called from any thread. This function is **not preemptable**.

Parameters

- `pInstanceData`: A pointer to the global service instance data. This pointer may be `NULL`.

B2.3.5.2 SINSTANCEGETDATA

```
void* SInstanceGetData( void )
```

The `SInstanceGetData` function retrieves the instance data pointer set by the service using the `SInstanceSetData` function.

This function can be called from any thread. This function is **not preemptable**.

Return Value

The value returned is the previously set pointer to the service instance data, or `NULL` if no instance data pointer has yet been set.

B2.3.6 Service Properties Functions

The service properties are written in the service manifest by the service developer. They can be accessed at runtime using the `SService` functions, which can be called from any thread.

B2.3.6.1 Service Properties

Service properties are divided into three categories:

- Framework configuration properties are interpreted by the implementation. They allow the service to configure the framework for its needs. A specific configuration property is the service identifier, which is a UUID.
- Service configuration properties that allow the service to define properties for its own configuration.
- Public properties that the service can define to communicate properties to its clients. For example this can be the service name, vendor etc.

Only public properties are visible to clients connecting to the Service Manager (see Chapter B7, "*Service Manager Protocol*").

The following namespaces of properties are defined:

Table B2-7: SSDI: Service Property Namespaces

Namespace	Meaning
`config.s.*`	Framework configuration properties.
`config.* (except config.s.*)`	Service configuration properties.
all except `config.*`	Service public properties.

This specification defines the following framework configuration properties. If an optional property is not listed in the service meta-data, the system will automatically assign a default value to this property. A service can always retrieve the following properties through the service property functions.

Table B2-8: SSDI: Service Configuration Properties

Property Name	Type	Meaning	Default
`config.s.serviceID`	UUID	The identifier of the service. This property is **mandatory**.	n/a
`config.s.multi_instance`	Boolean	Whether the system shall create a separate instance for each client session	`false`
`config.s.multi_session`	Boolean	Whether the instances of the service support multiple sessions. This property is ignored for multi-instance services.	`false`
`config.s.multi_command`	Boolean	Whether the sessions of the service support multiple simultaneous commands	`false`
`config.s.lazy_instantiation`	Boolean	Whether the service instance must be created for the first client session and destroyed when the last client session is closed. If `false`, it means that the service instance is created at boot-time, and destroyed when the Secure World is shutdown. This property is ignored for multi-instance	`false`

		services.	
`config.s.heap_size`	Integer	The size in bytes of the heap allocated for each instance. This size includes an overhead put by the implementation and that is implementation-dependent. Refer to the *Product Reference Manual* for approximation of this overhead. When the `SRVXCreate` entry point is called, it is guaranteed that `config.s.heap_size` bytes are entirely reserved for the process: the process will not start if such a heap cannot be allocated.	`16384` (16KB)
`config.s.heap_size.max`	Integer	The maximum size of the heap for the process. Allocations beyond this limit will always fail. Allocations between `config.s.heap_size` and `cofig.s.heap_size.max` may require that the system allocate new memory pages to the process, and may therefore fail.	Value of `config.s.heap_size`
`config.s.default_stack_size`	Integer	The default stack size in bytes. Threads created by the system for the service entry points will always have this stack size. Service created threads may have a specified stack size. Note that this stack size is entirely available to the service thread, and the stack will not contain any overhead due to system owned data.	`2048` (2KB)
`config.s.service_manager.allow_manager_mode`	Boolean	Whether the service is allowed to access the service manager in manager mode.	`false`

B2.3.6.2 SServiceGetAllProperties

```
S_RESULT SServiceGetAllProperties(
       OUT S_PROPERTY** ppProperties,
       OUT uint32_t*    pnPropertiesCount )
```

This function returns the list of all properties associated with the service.

The implementation allocates a single memory block containing an array of `S_PROPERTY` structures and the corresponding strings. Each element of the array contains a pointer to a single property's name and value. The name and value are zero-terminated Unicode strings encoded in UTF-8. These strings are part of the memory block allocated by the implementation.

Upon successful completion, the calling service is responsible for calling the `SMemFree` function to deallocate the properties block. Calling `SMemFree` will free the property array as well as each property’s name and value strings.

This function is **not preemptable**.

Parameters

- `ppProperties`: A pointer to a variable that will receive a pointer to the array of properties. The variable is set to `NULL` upon error.
- `pnPropertiesCount`: A pointer to a variable that will receive the number of elements in the returned array. The variable is set to zero upon error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.

Panic Reasons

This function may raise a panic for the following reasons:

- `ppProperties` or `pnPropertiesCount` is `NULL`.

B2.3.6.3 SServiceGetProperty

```
S_RESULT SServiceGetProperty(
        IN const char* pName,
       OUT char**      ppValue )
```

The `SServiceGetProperty` function retrieves a single service property

The implementation allocates the string returned. The caller must deallocate this string with `SMemFree`.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the string containing the name of the property to retrieve. This pointer must not be `NULL`
- `ppValue`: A pointer to the variable which will contain pointer to the property value string on success. This variable will be set to `NULL` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.

Panic Reasons

This function may raise a panic for the following reasons:

- `ppValue` is `NULL`.
- `pName` is `NULL`

B2.3.6.4 SServiceGetPropertyAsInt

```
S_RESULT SServiceGetPropertyAsInt (
        IN const char* pName,
       OUT uint32_t*   pnValue )
```

The `SServiceGetPropertyAsInt` function retrieves a single service property and converts it to an integer.

The syntax for integer properties is specified in section B2.3.2.1.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the string containing the name of the property to retrieve. This pointer must not be `NULL`
- `pnValue`: A pointer to the variable that will contain the value of the property on success, or zero on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.
- `S_ERROR_BAD_FORMAT` if the property value cannot be converted to a positive integer.

Panic Reasons

This function may raise a panic for the following reasons:

- `pnValue` is `NULL`.
- `pName` is `NULL`

B2.3.6.5 SSERVICEGETPROPERTYASBOOL

```
S_RESULT SServiceGetPropertyAsBool(
        IN const char* pName,
       OUT bool*       pbValue )
```

The `SServiceGetPropertyAsBool` function retrieves a single service property and converts it to a Boolean.

The syntax for boolean properties is specified in section B2.3.2.2.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the string containing the name of the property to retrieve. This pointer must not be `NULL`.
- `pbValue`: A pointer to the variable that will contain the value of the property on success or `false` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.
- `S_ERROR_BAD_FORMAT` if the property value cannot be converted to a boolean

Panic Reasons

This function may raise a panic for the following reasons:

- `pnValue` is `NULL`.
- `pName` is `NULL`

B2.3.7 SESSION FUNCTIONS

These functions can only be called from session threads (see section B2.2.4, “*Processes and Threads*”).

There are used to access the properties associated with the client that has opened the current client session.

B2.3.7.1 Client Information

The client UUID can be retrieved by calling the function `SSessionGetClientID`. This information is **not** accessible as a property.

The functions `SSessionGetAllClientProperties`, `SSessionGetClientProperty`, `SSessionGetClientPropertyAsInt`, and `SSessionGetClientPropertyAsBool` can be used to retrieve one or several properties of the client, and optionally convert them in an integer or a boolean.

The client identifier and the client properties that the service can retrieve depend on the nature of the client and the method it has used to connect:

Table B2-9: SSDI: Client Properties

Client Login type	**Client Identifier (returned by `SSessionGetClientID`)**	**Value of the `sm.client.login` property**	**Other Client Properties**
`TEEC_LOGIN_PUBLIC`	Nil	`"0x00000000"` (`S_LOGIN_PUBLIC`)	None
`TEEC_LOGIN_PRIVILEGED`		`"0x80000002"` (`S_LOGIN_PRIVILEGED`)	
`TEEC_LOGIN_USER`	A UUID determined by the Normal World Operating System	`"0x00000001"` (`S_LOGIN_USER`)	
`TEEC_LOGIN_GROUP`		`"0x00000002"` (`S_LOGIN_GROUP`)	
`TEEC_LOGIN_APPLICATION`		`"0x00000004"` (`S_LOGIN_APPLICATION`)	
`TEEC_LOGIN_USER_APPLICATION`		`"0x00000005"` (`S_LOGIN_APPLICATION_USER`)	
`TEEC_LOGIN_GROUP_APPLICATION`		`"0x00000006"` (`S_LOGIN_APPLICATION_GROUP`)	
`TEEC_LOGIN_AUTHENTICATION`	The property `sm.client.id` of the client manifest. See section B3.3.6.2, "*Manifest File Format*" for more details.	`"0x80000000"` (`S_LOGIN_AUTHENTICATION`)	All the properties in the client manifest, except `sm.client.id`, but including `sm.client.hash`
Secure World service	The calling service UUID	`"0xF0000000"` (`S_LOGIN_CLIENT_IS_SERVICE`)	All public properties of the calling service, i.e., the properties outside the `config.*` namespace

The property `sm.client.login` is typically retrieved as an integer using the function `SSessionGetClientPropertyAsInt`.

B2.3.7.2 SSessionGetClientID

```
void SSessionGetClientID( S_UUID* pClientID )
```

The `SSessionGetClientID` function retrieves the client identifier for a given session.

If the client is logged in as using the `S_LOGIN_PUBLIC` or `S_LOGIN_PRIVILEGED` mode it has no unique identity, and this function will return the Nil UUID as defined in **[RFC4122]**.

This function is **not preemptable**.

Parameters

- `pClientID`: A pointer to the UUID structure which will be filled with the client identifier.

Panic Reasons

This function may raise a panic for the following reasons:

- The function is called from a thread that is not a session thread.
- `pClientID` is NULL.

B2.3.7.3 SSessionGetAllClientProperties

```
S_RESULT SSessionGetAllClientProperties(
       OUT uint32_t*     pnPropertyCount,
       OUT S_PROPERTY** ppPropertyArray )
```

This function retrieves a list of all of the client properties.

The system allocates a single memory block containing an array of `S_PROPERTY` structures and the corresponding strings. Each element of the array contains a pointer to a single property's name and value. The name and value are zero-terminated Unicode strings encoded in UTF-8. These strings are part of the memory block allocated by the implementation.

If this operation is successful, the calling service is responsible for calling the `SMemFree` function to deallocate the block containing the properties. Calling `SMemFree` will free the property array as well as each property's name and value strings.

See the section B2.3.7.1, "*Client Information*" for a definition of the properties that can be retrieved using this function.

This function is **not preemptable**.

Parameters

- `pnPropertyCount`: A pointer to a variable which will be set with the number of properties. It is set to `NULL` upon error.
- `ppPropertyArray`: A pointer to a variable which will be set with a pointer to the property list. It is set to `NULL` upon error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.

Panic Reasons

This function may raise a panic for the following reasons:

- The function is called from a thread that is not a session thread.
- `pnPropertyCount` is `NULL`.
- `ppPropertyCount` is `NULL`.

B2.3.7.4 SSessionGetClientProperty

```
S_RESULT SSessionGetClientProperty(
         IN const char* pName,
        OUT char**      ppValue )
```

The `SSessionGetClientProperty` function returns the value of the client property whose name is given by `pName`.

`pName` must be a UTF-8 zero-terminated string.

The system allocates the memory for the string returned. On successful completion, the caller is responsible for deallocating this string with `SMemFree`.

See the section B2.3.7.1, "*Client Information*" for a definition of the properties that can be retrieved using this function.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the string containing the name of the property to retrieve.
- `ppValue`: A pointer to a variable which will contain a pointer to the UTF-8 zero-terminated value string on success, or `NULL` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.

Panic Reasons

This function may raise a panic for the following reasons:

- `pName` is `NULL`.
- `ppValue` is `NULL`.
- The function is called from a thread that is not a session thread.

B2.3.7.5 SSessionGetClientPropertyAsInt

```
S_RESULT SSessionGetClientPropertyAsInt(
         IN const char* pName,
        OUT uint32_t*   pnValue )
```

The `SSessionGetClientPropertyAsInt` function retrieves a single client property and converts it to an integer.

The syntax for integer properties is specified in section B2.3.2.1.

`pName` must be a UTF-8 zero-terminated string.

See the section B2.3.7.1, "*Client Information*" for a definition of the properties that can be retrieved using this function.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the string containing the name of the property to retrieve.
- `pnValue`: A pointer to the variable which will contain the value of the property on success, or zero on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.
- `S_ERROR_BAD_FORMAT` if the property value cannot be converted to a positive integer.

Panic Reasons

This function may raise a panic for the following reasons:

- `pName` is `NULL`.
- `pnValue` is `NULL`.
- The function is called from a thread that is not a session thread.

B2.3.7.6 SSessionGetClientPropertyAsBool

```
S_RESULT SSessionGetClientPropertyAsBool (
         IN const char* pName,
        OUT bool*        pbValue )
```

The `SSessionGetClientPropertyAsBool` function retrieves a single client property and converts it to a Boolean.

The syntax for boolean properties is specified in section B2.3.2.2.

See the section B2.3.7.1, "*Client Information*" for a definition of the properties that can be retrieved using this function.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the string containing the name of the property to retrieve.
- `pbValue`: A pointer to the variable that will contain the value of the property on success, or `false` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.
- `S_ERROR_BAD_FORMAT` if the property cannot be converted to a boolean

Panic Reasons

This function may raise a panic for the following reasons:

- `pnValue` is `NULL`.
- The function is called from a thread that is not a session thread.

B2.3.8 Memory Management API

These functions must be used by the service to manage its memory heap.

Note that the details of heap implementation are not specified at this level, and may vary between different versions of the software. In particular, the following behaviors are implementation defined:

- The contents of any newly allocated memory.
- The order and contiguity of buffers allocated by successive calls to `SMemAlloc` or `SMemRealloc`.

B2.3.8.1 SMemAlloc

```
void* SMemAlloc( uint32_t nSize )
```

This function allocates space for an object whose size in bytes is specified by `nSize`.

The pointer returned points to lowest byte address of the allocated space, and is guaranteed to be aligned such that it may be assigned as a pointer to any of the basic C types.

- For the ARM target implementations this means that this pointer is guaranteed to be at least 8-byte aligned.

If the space cannot be allocated, a `NULL` pointer is returned.

If the size of the space requested is zero, the value returned is still a non-`NULL` pointer that the service must not attempt to access.

This function is **not preemptable**.

Parameter

- `nSize`: The size of the buffer to be allocated.

Return Value

Upon successful completion, with size not equal to zero, the function returns a pointer to the allocated space. Otherwise, a `NULL` pointer is returned.

B2.3.8.2 SMemRealloc

```
void* SMemRealloc(
         IN void*    pBuff,
            uint32_t nNewSize )
```

This function changes the size of the memory object pointed to by `pBuff` to the size specified by `nNewSize`.

The content of the object remains unchanged up to the lesser of the new and old sizes, anything space in excess of the old size contains unspecified content.

If the new size of the memory object requires movement of the object, the space for the previous instantiation of the object is deallocated. If the space cannot be allocated, the original object remains allocated, and this function returns a `NULL` pointer.

If `pBuff` is `NULL`, `SMemRealloc` is equivalent to `SMemAlloc` for the specified size.

If `pBuff` does not match a pointer returned earlier `SMemAlloc` or `SMemRealloc` or if the space has previously been deallocated by a call to `SMemFree` or `SMemRealloc`, then the behavior is undefined.

The pointer returned obeys the same rules as pointers allocated via `SMemAlloc`.

If the memory block was allocated with `SMemAllocEx` from a memory pool different from 0, there is no guarantee that the size of the blocks allocated using `SMemAllocEx` can be changed using `SMemRealloc`. If reallocation is not supported for a given memory pool, the function `SMemRealloc` will always fail to extend the size of the block and will always silently ignore a request to decrease the size of a block.

This function is **not preemptable**.

Parameters

- `pBuff`: the pointer to the object to be reallocated.
- `nNewSize`: the new size required for the object.

Return Value

Upon successful completion, `SMemRealloc` returns a pointer to the (possibly moved) allocated space.

If there is not enough available memory, `SMemRealloc` returns a null pointer.

B2.3.8.3 SMemAllocEx

```
void* SMemAllocEx(uint32_t nPoolID, uint32_t nSize )
```

Allocates a block of memory from a given "memory pool". The precise definition of memory pools is implementation-defined. However, any implementation is required to support at least two memory pools:

- An allocation in the memory pool `0` must behave exactly like with `SMemAlloc`, i.e., `SMemAllocEx(0, nSize)` is equivalent to `SMemAlloc(nSize)`;
- The memory pool `1` should be preferred for the allocation of large content blocks (like video frames). In some implementations, this pool may use a dedicated method of allocation outside the heap of the service. Some other implementations will simply redirect these allocations in the service heap.

Except for the memory pool `0`, there is no guarantee that the size of the blocks allocated using `SMemAllocEx` can be changed using `SMemRealloc`. If reallocation is not supported for a given memory pool, the function `SMemRealloc` will always fail to extend the size of the block and will always silently ignore a request to decrease the size of a block.

Other memory pools may be available in some implementations.

Parameters

- `nPoolID`: the identifier of the memory pool to allocate from
- `nSize`: The size in bytes of the buffer to be allocated.

Return Value

Upon successful completion, the function returns a pointer to the allocated space. Otherwise, a `NULL` pointer is returned.

B2.3.8.4 SMEMFREE

```
void SMemFree( IN void *pBuff )
```

The `SMemFree` function causes the space pointed to by `pBuff` to be deallocated; that is, made available for further allocation.

If `pBuff` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `SMemAlloc` or `SMemRealloc`, or if the space has been deallocated by a call to `SMemFree` or `SMemRealloc`, the behavior is undefined.

This function is **not preemptable**.

Parameter

- `pBuff`: The pointer to the memory block to be freed.

B2.3.8.5 SMemMove

```
void* SMemMove(
        IN void*       pDest,
        IN const void* pSrc,
           uint32_t    nSize );
```

The `SMemMove` function copies `nSize` bytes from the object pointed to by `pSrc` into the object pointed to by `pDest`.

Copying takes place as if the `nSize` bytes from the object pointed to by `pDest` are first copied into a temporary array of `nSize` bytes that does not overlap the objects pointed to by `pDest` and `pSrc`, and then the `nSize` bytes from the temporary array are copied into the object pointed to by `pDest`.

This function is **not preemptable**.

Parameters

- `pDest`: A pointer to the destination buffer.
- `pSrc`: A pointer to the source buffer.
- `nSize`: The number of bytes to be copied.

Return Value

This function always returns `pDest`.

Future implementations reserve the right to return `NULL` to indicate an error.

B2.3.8.6 SMEMCOMPARE

```
int32_t SMemCompare(
       IN const void* pBuff1,
       IN const void* pBuff2,
          uint32_t    nSize);
```

The SMemCompare function compares the first nSize bytes of the object pointed to by pBuff1 to the first nSize bytes of the object pointed to by pBuff2. The first bytes in a buffer are those with the lowest address.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type uint8_t) that differ in the objects being compared.

This function is **not preemptable**.

Parameters

- pBuff1: A pointer to the first buffer.
- pBuff2: A pointer to the second buffer.
- nSize: The number of bytes to be compared.

Return Value

The SMemCompare function returns an integer greater than, equal to, or less than zero.

- If the first byte that differs is higher in pBuff1, then return an integer greater than zero.
- If the first nSize bytes of the two buffers are identical, return zero.
- If the first byte that differs is higher in pBuff2, then return an integer lower than zero.

B2.3.8.7 SMEMFILL

```
void* SMemFill(
        IN void*     pBuff,
           uint32_t  nByte,
           uint32_t  nSize);
```

The `SMemFill` function copies `nByte` bytes (converted to a `uint8_t`) into the first `nSize` bytes of the object pointed to by `pBuff`.

This function is **not preemptable**.

Parameters

- `pBuff`: A pointer to the destination buffer.
- `nByte`: The value to be set.
- `nSize`: The number of bytes to be set.

Return Value

The `SMemFill` function returns `pBuff`.

Future implementations reserve the right to return `NULL` to indicate an error.

B2.3.9 Traces and Debug Facilities

These functions and macros are useful for debugging purposes. They provide a general tracing facility and an assertion facility.

B2.3.9.1 SLogTrace, SLogWarning, SLogError

```
void SLogTrace( IN const char* pMessage, /* Args */ ...)
void SLogWarning( IN const char* pMessage, /* Args */ ...)
void SLogError( IN const char* pMessage, /* Args */ ...)
```

These functions send a message to an implementation-defined logging facility. The traces may be sent to a communication port, a system console, a file, etc.

The implementation visually differentiates the trace, warning and error messages.

Each call to one of these functions outputs one "log" line. The underlying trace system may add additional log information and add a new line after each log message. The precise content of the log line as well as the insertion of new lines depend on the underlying trace system. This is documented in the *Product Reference Manual*.

The transmitted message is generated from the message template, `pMessage`, and any subsequently following arguments.

In message template strings containing the `%` form of conversion, each conversion uses the first unused argument in the argument list. Variable conversion placeholders are denoted by the '`%`' character after which the following appear in sequence:

- Zero or more flags, in any order, which modify the meaning of the conversion specification.
  - '`-`': This flag indicates that the result of the conversion shall be left-justified within the field. If this flag is not specified the conversion will default to right-justified text.
  - '`+`': This flag indicates that the result of a signed conversion shall always begin with a sign ('`+`' or '`-`'). By default, when this flag isn't specified, the conversion only begins with a sign when a negative value is converted.
  - '` `' (`<space>`): This flag indicates that the first character of a signed conversion should be a '` `' if there is not a sign chracter. This means that if this flag and the '`+`' flag both appear, this flag shall be ignored. By default, no space will be added and positive results will appear one character earlier in the output message than the equivalent negative conversion.
  - '`0`': This flag indicates that `d`, `u`, `x`, and `X` conversion specifiers should be padded with leading zeros (following any indication of sign or base) instead of the default spaces. If the '`0`' and '`-`' flags both appear, or if a precision is specified, the '`0`' flag is ignored.
- An optional minimum field width. If the converted value has fewer bytes than the field width it shall be padded with spaces, by default on the left; but on the right if the left-adjustment flag ('`-`'), described above, is given. The field width takes the form of decimal digit.
- An optional precision that gives the minimum number of digits to appear for the `d`, `u`, `x`, and `X` conversion specifiers or the maximum number of bytes to be printed from a string in the `s` conversion specifiers.
  - The precision takes the form of a period ('`.`') followed by an optional decimal digit string, where a zero length digit string is treated as a numerical zero. If a precision appears with any other conversion specifier, it is ignored
- A conversion specifier character that indicates the type of conversion to be applied.
  - '`d`': The integer argument shall be converted to a signed decimal in the style "`[-]dddd`".
    - The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1.
    - The result of converting an integer with value of zero with an explicit precision of zero shall be no characters.

- ‘`u`’: The unsigned integer argument shall be converted to unsigned decimal format in the style "`dddd`".
  - The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1.
  - The result of converting zero with an explicit precision of zero shall be no characters.
- ‘`x`’: The unsigned integer argument shall be converted to unsigned hexadecimal format in the style "`dddd`"; the letters "`abcdef`" are used.
  - The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1.
  - The result of converting zero with an explicit precision of zero shall be no characters.
- ‘`X`’: Equivalent to the ‘`x`’ conversion specifier, except that letters "`ABCDEF`" are used instead of "`abcdef`"
- ‘`p`’: Equivalent to the conversion “`%08X`” with the argument being a `void*` pointer converted to a 32-bit integer
- ‘`c`’: The integer argument shall be converted to an unsigned char, and the resulting byte shall be written.
- ‘`s`’ The argument shall be a pointer to an array of characters. If this pointer is NULL, then the string “NULL” is displayed. Bytes from the array shall be written up to (but not including) any terminating NUL byte.
  - If the precision is specified, no more than that many bytes shall be written.
  - If the precision is not specified or is greater than the size of the array, the application must ensure that the array contains a NUL termination byte.
- ‘`%`’: there is no argument and the character ‘`%`’ is printed

Security Warning: *note that `pMessage` must never come from an untrusted source, such as a Normal World client. The trace functions may cause security violations if the message template string contains more conversion placeholders than there are arguments or if the conversion placeholder types differ from the arguments provided. Such security violations include tracing values in the stack or memory dereferenced by pointers in the stack.*

The correct and secure way to trace an arbitrary or untrusted string is to use “`SLogXXX("%s", pString)`”. The string is then guaranteed not to be expanded even if it contains conversion placeholders. In general, `pMessage` should almost always be a constant string literal.

Additionally, note that the conversion specifier must be one of the documented specifiers listed above. Using undocumented specifiers may result in the same security violations with values in the stack or pointer dereferenced being traced.

This function is implemented as a macro in the `ssdi.h` header file and is compiled out if the preprocessor symbol `SSDI_NO_TRACE` is defined. Note that, by default, traces are *enabled*.

This function is not preemptable.

Parameters

- `pMessage`: The formatted message template to log. If `pMessage` is equal to `NULL`, then the string “`NULL`” is displayed.
- Additional Arguments: The parameters to be used for expansion of the conversion specifiers in the template message.

B2.3.9.2 SAssert

```
void SAssert( <scalar expression> )
```

The `SAssert` macro inserts diagnostics into programs; it expands to a `void` expression. When it is executed, if `expression` (which shall have a scalar type) is `false` (that is, it compares equal to zero), `SAssert` logs the information about the particular call that failed and causes a panic.

The information written about the call that failed includes the text of the argument, the name of the source file, the source file line number, and the name of the enclosing function;

This macro is defined only if the name `SSDI_DEBUG` is defined; either from the compiler command line or with the preprocessor control statement “`#define SSDI_DEBUG`” before including `ssdi.h`. Note that, by default, assertions are not activated.

B2.3.10 THREAD API

The Thread API contains functions to create and manage threads. Threads can either be created by the system when a service entry point is called or created by the service itself.

B2.3.10.1 STHREADCREATE

```
S_RESULT SThreadCreate(
       OUT S_HANDLE* phThread,
           uint32_t  nStackSize,
           uint32_t  (*pEntryPoint)(void*),
        IN void*     pThreadArg )
```

This function creates a new thread, which runs in the current service instance.

When the newly-created thread starts the system calls the function `pEntryPoint` with `pThreadArg` as the single function argument. This allows the current thread to pass some data to t he new thread. Note that threads share a common memory space, so objects allocated in the instance heap or in a thread stack can be passed to the created thread. As with all multi-threaded code care must be taken to ensure that data passed using the heap , or more critically when using the stacks, will remain valid until the created thread no longer needs the data.

The new thread is completely detached from the calling thread. If the calling thread terminates, the new thread continues its execution until it returns from its entry point. The new thread is not considered as a session thread even if it is created by a session thread.

When `phThread` is non-`NULL` this function returns a thread handle that refers to the newly created thread. This handle is used when calling the functions `SThreadJoin` and `SThreadCancel`. Once the thread handle is no longer required it must be closed by calling the generic function `SHandleClose`, or some underlying resources will leak.

Note that the thread handle is a separate entity to the thread itself. The thread handle may exist after the thread has closed and a thread can exist without a thread handle.

The `nStackSize` parameter provides the size of the stack in bytes. If it is zero the default stack size, specified in the service properties, is used (see section B2.3.6.1, "*Service Properties*"). Note this stack size indicates the size that is entirely available to the service thread code; the system places no additional data into the stack prior to calling the thread entry point.

This function is **not preemptable**. The new thread will start its execution only when the current thread is preempted.

Parameters

- `phThread`: A pointer to a handle valued with the new thread handle. If this parameter is `NULL`, no thread handle is returned, but the thread is still created.
- `nStackSize`: The stack size of the new thread. If zero, the default stack size is used.
- `pEntryPoint`: A function pointer to the thread entry point.
- `pThreadArg`: A pointer to the thread argument object. This pointer is passed as an argument to the function `pEntryPoint`, and may be `NULL`.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to allocate the thread object or t he stack.

Panic Reasons

This function raises a panic for the following reasons:

- `pEntryPoint` is `NULL`.

B2.3.10.2 SThreadJoin

```
S_RESULT SThreadJoin(
           S_HANDLE       hThread,
       OUT uint32_t*      pnExitCode,
           S_TIME_LIMIT* pTimeLimit );
```

This function suspends the execution of the calling thread until the target thread terminates. It optionally returns the target thread's exit code. When this function returns successfully, it is guaranteed that the target thread has terminated.

This function is **preemptable**. While it is executing, another thread in the service instance may be scheduled.

This function is **cancellable**. When cancellation requests are not masked, this function will return the error code `S_ERROR_CANCEL` when the calling thread is cancelled either before calling `SThreadJoin` or during its execution.

Parameters

- `hThread`: The handle of the target thread.
- `pnExitCode`: If non-`NULL`, this pointer is filled with the target threads exit code. On error, the exit code is set to zero.
- `pTimeLimit`: The time limit after which the operation is aborted with the error code `S_ERROR_TIMEOUT`. If it is to `NULL`, the function blocks indefinitely. If `pTimeLimit` is not `NULL`, then *`pTimeLimit` must have been created by calling `STimeGetLimit`.
    - The timeout may be longer than requested due to scheduling activity of other processes and threads within the system.

Return Value

On successful completion, this function returns `S_SUCCESS`. Otherwise, one of the following error codes is returned:

- `S_ERROR_TIMEOUT`: The timeout has elapsed and the target thread has not yet been terminated.
- `S_ERROR_CANCEL`: The calling thread has been cancelled and the target thread has not yet been terminated.

Panic Reasons

This function raises a panic for the following reasons:

- `hThread` doesn't reference an existing thread handle.

B2.3.10.3 SThreadYield

```
void SThreadYield( void );
```

This function suspends the calling thread and lets another thread of the instance take the control.

By nature, this function is **preemptable**.

B2.3.10.4 STHREADSLEEP

```
S_RESULT SThreadSleep( IN const S_TIME_LIMIT* pTimeLimit )
```

This function causes the calling thread to be suspended until the specified time limit. The suspension time may be longer than requested due to the scheduling of other tasks by the system.

`pTimeLimit` can be `NULL` or `*pTimeLimit` must have been created by calling `STimeGetLimit`.

The function `SThreadSleep` is **preemptable**. It is also **cancellable**.

Parameters

- `pTimeLimit`: The time limit. If set to `NULL`, the function sleeps indefinitely. This is useful if the calling thread must wait until it is cancelled.

Return Value

- `S_SUCCESS` upon success.
- `S_ERROR_CANCEL` if the calling thread has been cancelled.

B2.3.10.5 SThreadCancel

```
void SThreadCancel(
           S_HANDLE hThread,
           uint32_t nReserved )
```

This function sends a cancellation request to the target thread.

If the target thread is already in a cancelled state this function does nothing; otherwise it sets the cancellation state of the target thread. The target thread can test if it is cancelled by calling the function `SThreadIsCancelled`. If the target thread is currently calling a cancellable function and if it has not masked cancellations, then the function may be cancelled. The effect of cancelling a function depends on the function and is described individually in the documentation of each function. Usually, the effect is to return the error code `S_ERROR_CANCEL`.

This function does not provide a guarantee that the target thread being cancelled is destroyed by the time that this function returns. It is the responsibility of the thread being cancelled to regularly test its cancellation state using the `SThreadIsCancelled` function and take appropriate action. Consequently, the target thread is also able to ignore the cancellation request.

This function is **not preemptable**. It only sends the cancellation request but does not schedule the target thread.

Parameters

- `hThread`: The handle of the thread to cancel.
- `nReserved`: Reserved for future use. Should be set to zero.

Panic Reasons

This function may raise a panic for the following reasons:

- `hThread` doesn't reference an existing thread handle.

B2.3.10.6 SThreadIsCancelled

```
bool SThreadIsCancelled( void* pReserved )
```

This function determines if the calling thread has been cancelled and optionally returns the cancellation code.

This function is **not preemptable**.

Parameters

- `pReserved`: reserved for future use. Should be set to `NULL`.

Return Value

- `true` if the thread is cancelled.
- `false` if the thread is not cancelled.

B2.3.10.7 SThreadResetCancel

```
void SThreadResetCancel( void )
```

This function resets the cancellation state of the calling thread. After this function has been called, the thread is no longer cancelled.

This function is **not preemptable**.

B2.3.10.8 SThreadMaskCancellation

```
void SThreadMaskCancellation( bool bMask )
```

This function controls the effects of cancellation requests on the cancellable functions called by the current thread.

If a thread has been set in to the cancelled state and cancellation requests are masked the thread is able to call a cancellable function as if the cancelled state was not set. The call to the cancellable function will not be interrupted and will terminate its execution normally.

If cancellation requests are not masked the cancellable function will be interrupted and usually return the error code `S_ERROR_CANCEL`.

Even if cancellation requests are masked a thread can still determine if it is cancelled by calling the function `SThreadIsCancelled`.

New threads that are created by the system default to having cancellation requests masked. This mask can be disabled by calling this function with `bMask` set to `false`, and be re-enabled again by calling this function with `bMask` set to `true`.

This function is **not preemptable**.

Parameters

- `bMask`: Boolean `true` if cancellation requests are to be masked, `false` otherwise.

B2.3.11 SEMAPHORE API

The Semaphore API defines semaphore objects and the associated operations.

B2.3.11.1 SSEMAPHORECREATE

```
S_RESULT SSemaphoreCreate(
           uint32_t  nInitialCount,
       OUT S_HANDLE* phSemaphore )
```

This function creates a new semaphore with an initial value of `nInitialCount`. This semaphore remains usable until the semaphore handle is closed using the generic `SHandleClose` function.

This function is **not preemptable**.

Parameters

- `nInitialCount`: The initial value of the semaphore.
- `phSemaphore`: A pointer to a variable that receives the value of the semaphore handle. Upon error this value is set to `S_HANDLE_NULL`.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.

Panic Reasons

- This function may raise a panic for the following reasons:
- `phSemaphore` is `NULL`.

B2.3.11.2 SSemaphoreAcquire

```
S_RESULT SSemaphoreAcquire(
            S_HANDLE            hSemaphore,
         IN const S_TIME_LIMIT* pTimeLimit )
```

This function is used by a calling process to acquire the semaphore. If the semaphore count value is already at zero, then the calling thread is suspended until some other thread increments the semaphore or the timeout elapses. If the semaphore count is not zero the count is decremented, and the function returns `S_SUCCESS`.

If `pTimeLimit` is `NULL`, there is no time-out: the operation will only complete when the semaphore is successfully acquired or it is cancelled.

If `pTimeLimit` is not `NULL`, it must have been created with the function `STimeGetLimit`. If `pTimeLimit` has been created with a zero time-out, the function returns immediately without yielding its control of the processor with the result `S_SUCCESS` if it can acquire the semaphore or with the error `S_ERROR_TIMEOUT` if it cannot.

This function is **preemptable** and **cancellable**.

Parameters

- `hSemaphore`: The semaphore handle.
- `pTimeLimit`: The time limit after which the operation is aborted with the error code `S_ERROR_TIMEOUT`. If it is `NULL`, there is no time-out and the function waits indefinitely until the semaphore is acquired or the function cancelled. If `pTimeLimit` is not `NULL`, then *`pTimeLimit` must have been created by calling `STimeGetLimit`.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_CANCEL` if the acquire operation has been cancelled before the semaphore has been acquired.
- `S_ERROR_TIMEOUT` if the attempt to lock the semaphore timed out.

Panic Reasons

This function may raise a panic for the following reasons:

- `hSemaphore` doesn't reference an existing semaphore handle.

B2.3.11.3 SSemaphoreRelease

```
void SSemaphoreRelease( S_HANDLE hSemaphore )
```

This function releases the semaphore, incrementing its count value. If some threads are waiting on this semaphore one of those threads becomes eligible for execution.

Note that the newly eligible thread will only be scheduled once the calling thread has yielded, and only then when it is scheduled by the system scheduler.

This function is **not preemptable**.

Parameters

- `hSemaphore`: The semaphore handle.

Panic Reasons

This function may raise a panic for the following reasons:

- `hSemaphore` doesn't reference an existing semaphore.

B2.3.12 FILE SYSTEM API

Each service has its own file-system. This file-system is shared between all the instances of the service. The file-system is **flat**: there is no directory structure within it.

B2.3.12.1 SFILEOPEN

```
S_RESULT SFileOpen(
           uint32_t    nStorageID,
        IN const char* pFilename,
           uint32_t    nFlags,
           uint32_t    nReserved,
       OUT S_HANDLE*   phFile);
```

The `SFileOpen` function creates or opens a file. It returns a handle that can be used to access the file.

The `nStorageID` parameter indicates which storage space to access. Possible values are:

- `S_FILE_STORAGE_PRIVATE`: This refers the private store of the current service. This storage space is accessible to all instances of this service;

The `nFlags` parameter is a set of flags that controls the access rights, sharing permissions, and file creation mechanism with which the file handle is opened. The value of the `nFlags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- Access control flags:
  - `S_FILE_FLAG_ACCESS_READ`: The file is opened with the read access right. This allows the service to call the function `SFileRead`.
  - `S_FILE_FLAG_ACCESS_WRITE`: The file is opened with the write access right. This allows the service to call the function `SFileWrite` and `SFileTruncate`.
  - `S_FILE_FLAG_ACCESS_WRITE_META`: The file is opened with the write-meta access right. This allows the service to call the functions `SFileCloseAndDelete` and `SFileRename`.
- Sharing permission control flags:
  - `S_FILE_FLAG_SHARE_READ`: The caller allows another handle on the file to be created with read access.
  - `S_FILE_FLAG_SHARE_WRITE`: The caller allows another handle on the file to be created with write access.
- File creation control flags:
  - `S_FILE_FLAG_CREATE`: If the specified file does not exist it is created and a handle returned on the new file. If the file already exists then it is simply opened.
  - `S_FILE_FLAG_EXCLUSIVE`: This flag is only meaningful if the `S_FILE_FLAG_CREATE` is also set, and is ignored otherwise. When both `S_FILE_FLAG_CREATE` and `S_FILE_FLAG_EXCLUSIVE` are set, the file is created only if it does not already exist. Otherwise, the error `S_ERROR_ACCESS_CONFLICT` is returned.

Multiple handles may be opened on the same file simultaneously, but sharing must be explicitly allowed. More precisely, at any one time the following constraints apply: if more than one handle are opened on the same file, and if any of this file handle was opened with the flag `S_FILE_FLAG_READ`, then all the file handles must have been opened with the flag `S_FILE_FLAG_SHARE_READ`. There is a similar constraint with the flags `S_FILE_FLAG_WRITE`/ `S_FILE_FLAG_SHARE_WRITE`. Accessing a file with write-meta rights is exclusive and can never be shared.

When the function `SFileOpen` is called and if opening the file would violate these constraints, then the function returns the error code `S_ERROR_ACCESS_CONFLICT`.

Any bits in `nFlags` not defined above are reserved for future use and must be set to zero. Otherwise, the error code `S_ERROR_BAD_PARAMETERS` is returned.

The following examples illustrate the behavior of the `SFileOpen` function when called twice on the same file. Note that for readability, the flag names used in the table have been abbreviated by removing the '`S_FILE_FLAG_`' prefix from their name, and any non-`S_SUCCESS` error codes have been shortened by removing the '`S_ERROR_`' prefix.

Table B2-10: SSDI: SFileOpen Sharing Rules

Value of `nFlags` for `SFileOpen` 1	Value of `nFlags` for `SFileOpen` 2	Return Code of `SFileOpen` 2	Comments
`READ`	`READ`	`ACCESS_CONFLICT`	The file handles have not been opened with the flag `FLAG_SHARE_READ`. Only the first call will succeed.
`READ | SHARE_READ`	`READ`	`ACCESS_CONFLICT`	Not all the file handles have been opened with the flag `FLAG_SHARE_READ`. Only the first call will succeed.
`READ | SHARE_READ`	`READ | SHARE_READ`	`S_SUCCESS`	All the file handles have been opened with the flag `FLAG_SHARE_READ`.
`READ`	`WRITE`	`ACCESS_CONFLICT`	Files handles are not opened with share flags. Only the first call will succeed.
`READ | SHARE_READ | SHARE_WRITE`	`WRITE | SHARE_READ | SHARE_WRITE`	`S_SUCCESS`	All the file handles have been opened with the share flags.
`READ | SHARE_READ | WRITE | SHARE_WRITE`	`WRITE_META`	`ACCESS_CONFLICT`	The write-meta flag indicates an exclusive access to the file. Only the first call will succeed.
`SHARE_READ`	`WRITE | SHARE_WRITE`	`ACCESS_CONFLICT`	A file can be opened with only share flags, which can be used to lock the access to a file against a given mode (here we prevent subsequent accesses in write mode).
`0`	`READ | SHARE_READ`	`ACCESS_CONFLICT`	A file can be opened with no flag set, which can be used to completely lock all subsequent attempts to access the file. Only the first call will succeed.

Each handle opened through the `SFileOpen` function has an associated file position indicator. When the file is opened, the position is initially set at the beginning of the file.

This function is **preemptable**.

Parameters

- `nStorageID`: This parameter determines the storage to use. It must be `S_FILE_STORAGE_PRIVATE`.
- `pFilename`: A pointer to the name of the file encoded as a zero-terminate string of bytes. The length of the string shall not exceed `S_FILE_NAME_MAX` bytes (excluding the zero-ending character). If the filename contains more than `S_FILE_NAME_MAX` bytes, the error `S_ERROR_BAD_PARAMETERS` is returned. Filenames must not contain the following bytes (otherwise the error `S_ERROR_BAD_PARAMETERS` is returned):
    - Control characters: char < 0x20
    - Separator character '/': char = 0x2F
    - Separator character '\': char = 0x5C
    - Control character: char ≥ 0x7F.
- `nFlags`: The flags which determine the settings under which the file is opened.
- `nReserved`: Reserved for future use. Should be set to zero.

- `phFile`: A pointer to the handle, which contains the opened handle upon successful completion. If this function fails for any reason, the value pointed to by `phFile` is set to `S_HANDLE_NULL`. When the file handle is no longer required, it must be closed using a call to the generic `SHandleClose` function.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ACCESS_CONFLICT` if an access right conflict was detected while opening the file.
- `S_ERROR_BAD_PARAMETERS` if the filename is too long, contains invalid characters or if the value of `nFlags` is not a valid combination of known flags.
- `S_ERROR_ITEM_NOT_FOUND` if the file cannot be found in the storage denoted by `nStorageID`.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_STORAGE_CORRUPTED` if the underlying storage is corrupted, either due to a physical failure or an attack.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `pFilename` is `NULL`.
- `phFile` is `NULL`.
- `nStorageID` is not `S_FILE_STORAGE_PRIVATE`.

B2.3.12.2 SFILEREAD

```
S_RESULT SFileRead(
            S_HANDLE  hFile,
        OUT uint8_t*  pBuffer,
            uint32_t  nSize,
        OUT uint32_t* pnCount )
```

This function attempts to read `nSize` bytes from the file associated with the handle `hFile` into the buffer pointed to by `pBuffer`.

The file handle must have been opened with the read access right.

The bytes are read starting at the position in the file indicated by the position associated with the file handle. The handle's file position is incremented by the number of bytes actually read.

On completion `SFileRead` sets the number of bytes actually read in the `uint32_t` pointed to by `pnCount`. The value written to `*pnCount` may be less than `nSize` if the number of bytes until the end-of-file is less than `nSize`. It is set to 0 if the file position is beyond the end-of-file. This is the only case where `*pnCount` may be less than `nSize`.

No data transfer can occur past the current end-of-file. If an attempt is made to read past the end-of-file, the `SFileRead` function stops reading data at the end-of-file and returns the data read up to that point. The file position indicator is then set at the end-of-file. If the file pointer is at, or past, the end of the file when this function is called this function does nothing, no bytes are copied to `*pBuffer` and `*pnCount` is set to 0.

This function is **preemptable**.

Parameters

- `hFile`: The file handle.
- `pBuffer`: A pointer to the memory which, upon successful completion, contains the bytes read.
- `nSize`: The number of bytes to read.
- `pnCount`: A pointer to the variable which upon successful completion contains the number of bytes read.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ACCESS_DENIED` if the file handle has not been opened with the read access right.
- `S_ERROR_STORAGE_CORRUPTED` if data corruption is detected.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.
- `pBuffer` is `NULL`.
- `pnCount` is `NULL`.

B2.3.12.3 SFileWrite

```
S_RESULT SFileWrite(
           S_HANDLE      hFile,
       IN const uint8_t* pBuffer,
           uint32_t      nSize )
```

The `SFileWrite` function writes `nSize` bytes from the buffer pointed to by `pBuffer` to the file associated with the open file handle `hFile`.

The file handle must have been opened with the write access permission.

If the current file position points before the end-of-file then `nSize` bytes are written to the file. If the current file position points beyond the file's end, then the file is first extended with zero bytes until the length indicated by the file position indicator is reached, and then `nSize` bytes are written to the file. The size of the file can be increased as a result of this operation.

The file-position indicator for the file is advanced by `nSize`. The file position indicators of other file handles opened on the same file are not changed.

Writing in a file is atomic; either the entire operation completes successfully or no write is done at all.

This function is **preemptable**.

Parameters

- `hFile`: The file handle.
- `pBuffer`: The buffer containing the data to be written.
- `nSize`: The number of bytes to write.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ACCESS_DENIED` if the file handle has not been opened with the write access right.
- `S_ERROR_STORAGE_CORRUPTED` if data corruption is detected.
- `S_ERROR_STORAGE_NO_SPACE` if insufficient storage space is available.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.
- `pBuffer` is `NULL`.

B2.3.12.4 SFileTruncate

```
S_RESULT SFileTruncate(
           S_HANDLE hFile,
           uint32_t nSize )
```

The function `SFileTruncate` changes the size of a file. If `nSize` is less than the current size of the file then all bytes beyond `nSize` are removed. If `nSize` is greater than the current size of the file then the file is extended by adding zero bytes at the end of the file.

The file handle must have been opened with the write access permission.

This operation does not change the file position of any handle opened on the file. Note that if the current file position of such a handle is beyond `nSize`, the file position will point beyond the file's end after truncation.

Truncating a file is atomic: either the file is successfully truncated or nothing happens.

This function is **preemptable**.

Parameters

- `hFile`: The file handle.
- `nSize`: The new size of the file.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ACCESS_DENIED` if the file handle has not been opened with the write access right.
- `S_ERROR_STORAGE_CORRUPTED` if data corruption is detected.
- `S_ERROR_STORAGE_NO_SPACE` if insufficient storage space is available to perform the operation.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.

B2.3.12.5 SFileSeek

```
S_RESULT SFileSeek(
           S_HANDLE hFile,
           int32_t  nOffset,
           S_WHENCE eWhence )
```

The `SFileSeek` function sets the file position indicator associated with the file handle.

The parameter `eWhence` controls the meaning of `nOffset`:

- If `eWhence` is `S_FILE_SEEK_SET`, the file position is set to `nOffset` bytes from the beginning of the file.
- If `eWhence` is `S_FILE_SEEK_CUR`, the file position is set to its current location plus `nOffset`.
- If `eWhence` is `S_FILE_SEEK_END`, the file position is set to the size of the file plus `nOffset`.

The `SFileSeek` function may be used to set the file position beyond the existing data in the file; this does not constitute an error. However, the file position indicator does have a maximum value which is `S_FILE_MAX_POSITION`. If the value of the file position indicator resulting from this operation would be greater than `S_FILE_MAX_POSITION`, the error `S_ERROR_OVERFLOW` is returned.

If an attempt is made to move the file position before the beginning of the file, the file position is set at the beginning of the file. This does not constitute an error.

This function is **not preemptable**.

Parameters

- `hFile`: The file handle.
- `nOffset`: The number of bytes to move the file position. A positive value moves the file position forward; a negative value moves the file position backward.
- `eWhence`: Selects the position in the file from which to calculate the new position.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OVERFLOW`: If the value of the file position indicator resulting from this operation would be greater than `S_FILE_MAX_POSITION`.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.
- `eWhence` is not `S_FILE_SEEK_SET`, `S_FILE_SEEK_CUR` or `S_FILE_SEEK_END`.

B2.3.12.6 SFILETELL

```
uint32_t SFileTell( S_HANDLE hFile )
```

The `SFileTell` function obtains the current position in the file for the specified file handle.

This function is **not preemptable**.

Parameters

- `hFile`: The file handle.

Return Value

The current position in the file, expressed as the number of bytes from the beginning of the file.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.

B2.3.12.7 SFileEOF

```
bool SFileEOF( S_HANDLE hFile )
```

The `SFileEOF` function determines if the current file position is at the end of the file.

This function is **not preemptable**.

Parameters

- `hFile`: the file handle.

Return Value

This function returns `true` if the file position is at the end of the file or beyond, `false` otherwise.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.

B2.3.12.8 SFileCloseAndDelete

```
S_RESULT SFileCloseAndDelete( S_HANDLE hFile )
```

The `SFileCloseAndDelete` function marks a file for deletion and closes the file handle.

The file handle must have been opened with the write-meta access right.

Deleting a file is atomic; once this function returns the file is definitely deleted and no more open handles for that file exist.

If `hFile` is `S_HANDLE_NULL`, the function does nothing and returns `S_SUCCESS`.

This function is **preemptable**.

Parameters

- `hFile`: The file handle.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ACCESS_DENIED` if the file handle has not been opened with the write-meta access right.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_STORAGE_CORRUPTED` if data corruption is detected.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file and is not `S_HANDLE_NULL`.

B2.3.12.9 SFileRename

```
S_RESULT SFileRename(
            S_HANDLE     hFile,
         IN const char*  pNewFilename )
```

The function `SFileRename` changes the name of a file. The file handle must have been opened with the write-meta access right.

Renaming a file is an atomic operation; either the file is renamed or nothing happens.

This function is **preemptable**.

Parameters

- `hFile`: The file handle.
- `pNewFilename`: The new file name encoded as a zero-terminated string of bytes. The length of the string shall not exceed `S_FILE_NAME_MAX` bytes (excluding the zero-ending character) and shall not contain the following bytes:
    - Control characters: char < 0x20
    - Separator character “`/`”: char = 0x2F
    - Separator character “`\`”: char = 0x5C
    - Control character: char ≥ 0x7F

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ACCESS_CONFLICT` if a file with the same name already exists.
- `S_ERROR_ACCESS_DENIED` if the file handle has not been opened with the write-meta access right.
- `S_ERROR_BAD_PARAMETERS` if `pNewFilename` is too long or contains illegal characters.
- `S_ERROR_STORAGE_CORRUPTED` if data corruption is detected.
- `S_ERROR_STORAGE_NO_SPACE` if insufficient storage space is available.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFile` does not reference an opened file.
- `pNewFilename` is `NULL`.

B2.3.12.10 SFileGetSize

```
S_RESULT SFileGetSize(
           uint32_t      nStorageID,
        IN const char*   pFilename,
       OUT uint32_t*     pnFileSize )
```

The `SFileGetSize` function returns the current size of a file. The file is referenced by its name. This function succeeds even if file handles are currently opened on the file.

This function is **preemptable**.

Parameters

- `nStorageID`: This parameter determines the storage where the file is found. Possible values are `S_FILE_STORAGE_PRIVATE`. See the function `SFileOpen` for more information.
- `pFilename`: The name of the file.
- `pnFileSize`: A pointer to the variable which contains the file size in bytes upon successful completion.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the file was not found in the storage.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_STORAGE_CORRUPTED` if the underlying storage is corrupted, either due to a physical failure or an attack.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `pnFileSize` is `NULL`.
- `pFilename` is `NULL`.
- `nStorageID` is not `S_FILE_STORAGE_PRIVATE`.

B2.3.12.11 SFileEnumerationStart

```
S_RESULT SFileEnumerationStart (
            uint32_t  nStorageID,
         IN char*     pFilenamePattern,
            uint32_t  nReserved1,
            uint32_t  nReserved2
        OUT S_HANDLE* phFileEnumeration )
```

The function `SFileEnumerationStart` starts an enumeration of a set of files. A set of files to be enumerated is defined as all the files whose name matches a given pattern. A pattern can contain the following wildcards:

- '`*`' is the wildcard for 0 or more characters.
- '`?`' is the wildcard for 0 or 1 character.

In order to specify '`*`' and '`?`' character in the pattern string (not interpreted as wildcards), these characters must be preceded by the escape '`\`' character. A '`\`' character that is not directly followed by '`*`' or '`?`' is forbidden.

The files are enumerated using the function `SFileEnumerationGetNext`.

The enumeration does not necessarily reflect a given consistent state of the storage: during the enumeration, other threads or other instances may create, delete, or rename files.

When the enumeration is no longer required, the handle must be destroyed using the generic `SHandleClose` function.

This function is **preemptable**.

Parameters

- `nStorageID`: The identifier of the storage in which the files must be enumerated. Possible values are:
  - `S_FILE_STORAGE_PRIVATE`: Indicates the storage is private to the service.
- `pFilenamePattern`: the filename pattern. This parameter may be NULL. In this case, all the files in the storage-space of the application are enumerated. The pattern shall not contain the following bytes:
  - Control characters: char < 0x20
  - Separator character '`/`': char = 0x2F
  - Separator character '`\`' (char = 0x5C) not directly followed by asterisk character '`*`' (char = 0x2A) or question-mark character '`?`' (char = 0x3F)
  - Control character: char ≥ 0x7F
- `nReserved1`, `nReserved2`: reserved for future use. Should be set to zero
- `phFileEnumeration`: a pointer filled with a handle on the file enumeration. Set to `S_HANDLE_NULL` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_BAD_PARAMETERS` if `pFilenamePattern` contains illegal characters.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_STORAGE_CORRUPTED` if the underlying storage is corrupted, either due to a physical failure or an attack.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `phFileEnumeration` is `NULL`.

- `nStorageID` is different from `S_FILE_STORAGE_PRIVATE`.

B2.3.12.12 SFileEnumerationGetNext

```
S_RESULT SFileEnumerationGetNext (
           S_HANDLE      hFileEnumeration,
       OUT S_FILE_INFO** ppFileInfo )
```

The function `SFileEnumerationGetNext` retrieves the information on the next file in the file enumeration.

If there are no more elements in the file enumeration, this function returns the error code `S_ERROR_ITEM_NOT_FOUND`.

This function is **preemptable**.

Parameters

- `hFileEnumeration`: a handle on the file enumeration.
- `ppFileInfo`: a pointer filled with a pointer on a structure `S_FILE_INFO` allocated by the API. This structure must be deallocated by the caller using `SMemFree`. This will deallocate the structure as well as the strings it contains.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if there are no more elements in the file enumeration.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_STORAGE_CORRUPTED` if the underlying storage is corrupted, either due to a physical failure or an attack.
- `S_ERROR_STORAGE_UNREACHABLE` if the underlying storage is unreachable.

Panic Reasons

This function may raise a panic for the following reasons:

- `hFileEnumeration` is not a valid handle on the file enumeration.
- `ppFileInfo` is `NULL`.

B2.3.13 DATE AND TIME MANAGEMENT API

Each service has its own date that can be set independently of the date of the other services. This date is expressed as a number of seconds since an arbitrary origin that is persistent and that is dependent on the service.

Services can also retrieve an absolute 'wall-clock' time, which may be used to initialize the service-specific date.

Conversion functions are also available to convert the time expressed as seconds into a Gregorian calendar representation.

Finally, the function `STimeGetLimit` can be used to fill-in a `S_TIME_LIMIT` structure with an absolute time limit. A time limit can be used to automatically cancel some functions after a certain timeout is reached.

B2.3.13.1 SDATESET

```
S_RESULT SDateSet (
           int32_t  nSeconds,
           uint32_t nReserved )
```

Sets the service's current date as the number of seconds since an arbitrary service-dependent origin.

Only the date for the current service is modified, not the date of other services. This will affect all the instances of the current service. The modification of the date is atomic and persistent across device reboots.

This function is **preemptable**.

Parameters

- `nSeconds`: the number of seconds since an arbitrary origin
- `nReserved`: reserved for future use. Must be set to zero.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if not enough memory is available to complete the operation.
- `S_ERROR_STORAGE_CORRUPTED` if the storage for this date source is detected to be corrupted, either due to a physical failure or an attack.
- `S_ERROR_STORAGE_NO_SPACE` if insufficient storage space is available to complete the operation.
- `S_ERROR_STORAGE_UNREACHABLE` if the storage for this date source is currently unavailable.

B2.3.13.2 SDateGet

```
S_RESULT SDateGet(
       OUT int32_t*  pnSeconds,
       OUT uint32_t* pnDateStatus,
       uint32_t      nReserved )
```

Retrieves the current date, expressed as the number of seconds since an arbitrary origin.

Upon succesfull completion the pnDateStatus output parameter indicates the current status of the service date source. It can have the following values:

- `S_DATE_STATUS_NOT_SET` means the current date has not been set. The service may set the date by calling the function `SDateSet`
- `S_DATE_STATUS_SET` means the current date is correctly set
- `S_DATE_STATUS_NEEDS_RESET` means the current date has been set but may have been corrupted and must no longer be trusted. In such a case it is recommended that the service resynchronize the date by calling the function SDateSet. Until the date has been reset, the status `S_DATE_STATUS_NEEDS_RESET` will be always returned

Initially the date status is `S_DATE_STATUS_NOT_SET`; and it will remain with this status until the service set the date by calling the function `SDateSet`. Once a service has synchronized the date the status can be `S_DATE_STATUS_SET` or `S_DATE_STATUS_NEEDS_RESET`. Once the status has become `S_DATE_STATUS_NEEDS_RESET` it will keep this status until the date is re-synchronized by calling `SDateSet`.

The following figure shows the state machine of the date status:

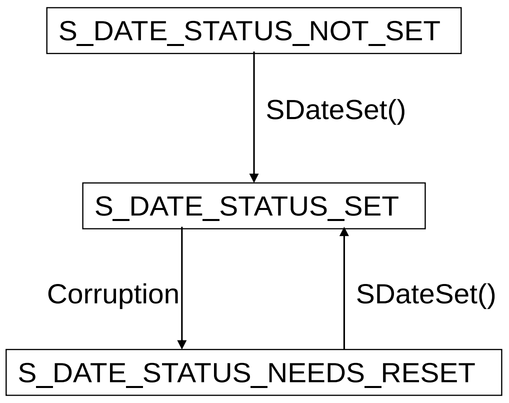


Figure B2-3. Date Status State Machine.

The meaning of the status `S_DATE_STATUS_NEEDS_RESET` depends on the level of protection level provided by the hardware implementation and the underlying real-time clock (RTC) driver. The following protection levels are defined:

- Level 1: the flag `S_DATE_STATUS_NEEDS_RESET` is set when the implementation detects a date rollback.
- Other protection levels depend on the device integration and on the properties of the underlying RTC. These additional protection levels are documented in the *Product Reference Manual*.

The protection level supported by an implementation is indicated by the following implementation property:

Property Name	Content type	Meaning
`date.protectionLevel`	Integer	Indicates the protection level provided by the date implementation.

The service can retrieve these properties by using the implementation property functions (see section B2.3.17).

When the status is set to S_DATE_NOT_SET, the value returned in pnSeconds is set to the value returned by the function SClockGet.

The value of the service date may exceed the range of the int32_t integer type. In this case, the function returns the error S_ERROR_OVERFLOW, but still computes the service date and date status as specified above, except that the number of seconds is truncated to 32 bits. For example, if the service sets the date to 2^{31}–100, then after 100 seconds, the service date is 2^{31}, which is not representable with an int32_t. In this case, the function SDateGet returns S_ERROR_OVERFLOW and sets *pnSeconds to –2174483647 (which is 2^{31} truncated to 32 bits or 0x80000000 in hexadecimal).

If the function SDateGet returns any other error but S_ERROR_OVERFLOW, then the status in pnDateStatus is set to S_DATE_STATUS_NOT_SET.

This function is **preemptable**.

Parameters

- pnSeconds: A pointer to the placeholder to be set with the current time. This placeholder is set to 0 for an error different from S_ERROR_OVERFLOW.
- pnDateStatus: A pointer to the placeholder to be set with the date status. This placeholder is set to S_DATE_STATUS_NOT_SET for an error different from S_ERROR_OVERFLOW.
- nReserved: reserved for future use. Should be set to zero.

Return Value

- S_SUCCESS in case of success.
- S_ERROR_OVERFLOW: the service date overflows the range of an int32_t. *pnSeconds is still set to the service date truncated to 32 bits (i.e. modulo 2^{32}) and the date status is set as described above.
- S_ERROR_OUT_OF_MEMORY if not enough memory is available to complete the operation.
- S_ERROR_STORAGE_CORRUPTED if the storage for this date source is detected to be corrupted, either due to a physical failure or an attack.
- S_ERROR_STORAGE_NO_SPACE if insufficient storage space is available to complete the operation.
- S_ERROR_STORAGE_UNREACHABLE if the storage for this date source is currently unavailable.

Panic Reasons

This function may raise a panic for the following reasons:

- pnSeconds is NULL.
- pnDateStatus is NULL.

B2.3.13.3 SClockGet

```
int32_t SClockGet();
```

Retrieves an absolute wall-clock time expressed as the number of seconds since midnight on January 1, 1970.

The value returned is a best-estimated time value and should not be considered as trusted, as it may be derived from a non-trusted time source such as the Normal World operating system. How this value is estimated is implementation-dependent.

This function is **not preemptable**.

Return Value

- The current wall-clock time expressed as the number of seconds since midnight on January 1, 1970.

B2.3.13.4 SDateConvertSecondsToCalendar

```
S_RESULT SDateConvertSecondsToCalendar(
        IN int32_t           nSeconds,
        IN const S_CALENDAR* pOrigin,
       OUT S_CALENDAR*       pDate )
```

Helper function to convert a number of seconds since a known origin into a calendar representation.

This function fills `pDate` with the date calculated as `pOrigin + nSeconds`.

`SDateConvertSecondsToCalendar` accepts arbitrary values of the fields in `pOrigin`; if field values are outside of real date ranges refer to `SDateConvertCalendarToSeconds` for an indication of how `pOrigin` will be converted to a second count. Note that the field `nDayOfWeek` is always ignored in `pOrigin`.

This function always returns a normalized date in `pDate`; that is `pDate` will always fall within real date ranges and `nDayOfWeek` will always be computed by the function.

This function is **not preemptable**.

Parameters

- `nSeconds`: a number of seconds.
- `pOrigin`: the date of the origin. The field `nDayOfWeek` is ignored.
- `pDate`: filled in the date calculated as `pOrigin + nSeconds`.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OVERFLOW` if the calculated value of `pOrigin + nSeconds` cannot be represented in a normalized `S_CALENDAR` structure. This happens only when the `nYear` field of the normalized `S_CALENDAR` structure is not in the range −2147483648 to +2147483647.

Panic Reasons

This function may raise a panic for the following reasons:

- `pOrigin` is NULL.
- `pDate` is NULL.

B2.3.13.5 SDateConvertCalendarToSeconds

```
S_RESULT SDateConvertCalendarToSeconds(
        IN const S_CALENDAR* pOrigin,
        IN const S_CALENDAR* pDate,
       OUT int32_t*          pnSeconds )
```

Helper function to calculate the difference in seconds between two dates expressed in calendar structures.

`SDateConvertCalendarToSeconds` accepts arbitrary values of the fields in `pOrigin` and `pDate` calendars. The field `nDayOfWeek` of both `pOrigin` and `pDate` is ignored.

The dates described by the calendars are converted to seconds for the year they describe as specified by the following pseudo-code, where the arithmetic is assumed to be done on signed integers of arbitrary precision:

```
S_CALENDAR sCal;

nSeconds  = MonthToSeconds(sCal.nMonth, sCal.nYear);
nSeconds += (sCal.nDay - 1) * 24 * 60 * 60;
nSeconds +=  sCal.nHour * 60 * 60;
nSeconds +=  sCal.nMinute * 60;
nSeconds +=  sCal.nSecond;

// At this point, the number of seconds since start of sCal.nYear is
nSeconds;

function MonthToSeconds(nMonth, nYear)
    while (nMonth > 12)
    {
       nDayCount += 365;
       if IsLeapYear(nYear) nDayCount++
       nMonth -= 12;
       nYear++;
    }
    while (nMonth <= 0)
    {
        nYear--;
        nDayCount -= 365;
        if IsLeapYear(nYear) nDayCount--;
        nMonth += 12;
    }

    if (nMonth >= 2)     nDayCount += 31;
    if (nMonth >= 3)
    {
        nDayCount += 28;
        if (IsLeapYear(nYear))   nDayCount++;
    }
    if (nMonth >= 4)     nDayCount += 31;
    if (nMonth >= 5)     nDayCount += 30;
    if (nMonth >= 6)     nDayCount += 31;
    if (nMonth >= 7)     nDayCount += 30;
    if (nMonth >= 8)     nDayCount += 31;
    if (nMonth >= 9)     nDayCount += 31;
    if (nMonth >= 10)    nDayCount += 30;
    if (nMonth >= 11)    nDayCount += 31;
    if (nMonth >= 12)    nDayCount += 30;
    return (nDayCount * 24 * 60 * 60);
}
function IsLeapYear(nYear)
```

```
{
    if ((nYear % 400) == 0)
         return true;  // This is a leap year.
    else if ((nYear % 100) == 0)
         return false; // This is not leap year.
    else if ((nYear % 4) == 0)
        return true;   // This is a leap year.
    else
        return false; // By default not a leap year.
}
```

Note that this function does not check that the range of the inputs in the calendar functions corresponds to valid ranges for a real date - for example `nDays` may exceed the actual number of days in `nMonth`. If values in the structure exceed valid ranges the function will still return a second count in accordance with the pseudo code above.

Parameters

- `pOrigin`: the date of the origin.
- `pDate`: the date to convert.
- `pnSeconds`: filled with the difference between `pDate` and `pOrigin` as a number of seconds.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OVERFLOW` if the difference between `pDate` and `pOrigin` cannot be represented in a signed 32-bit integer (i.e. is not in the range −2147483648 to +2147483647 seconds).

Panic Reasons

This function may raise a panic for the following reasons:

- `pOrigin` is `NULL`.
- `pDate` is `NULL`.
- `pnSeconds` is `NULL`.

B2.3.13.6 STimeGetLimit

```
void STimeGetLimit(
           uint32_t      nTimeout,
       OUT S_TIME_LIMIT* pTimeLimit )
```

Generates an absolute time limit from a relative timeout value.

The absolute time limit is equal to the current time plus the specified timeout.

This function is **not preemptable**.

A pointer to a `S_TIME_LIMIT` structure can be passed to the following functions:

- `SThreadSleep`
- `SThreadJoin`
- `SSemaphoreAcquire`
- `SXControlOpenClientSession`
- `SXControlInvokeCommand`

Note that any function that takes a `S_TIME_LIMIT*` parameter can be also be passed the `NULL` pointer to denote an infinite time-out.

Parameters

- `nTimeout`: The timeout in milliseconds, or the special value `S_TIMEOUT_INFINITE`.
- `pTimeLimit`: A pointer to a variable that will receive the generated `S_TIME_LIMIT` structure.

Panic Reasons

This function may raise a panic for the following reasons:

- `pTimeLimit` is `NULL`.

B2.3.14 CRYPTOGRAPHIC API

The Cryptographic API is specified in Chapter B4 "*Trusted Foundations* Cryptographic API" except for the non-standard functions that concerns the Cryptoki Update Shortcut features, which are specified in section B2.3.14.1 and following.

All the functions of the Cryptographic API are **preemptable** except `C_Login` and `C_Logout`, which are empty functions provided only for compatibility with the PKCS#11 standard. None are **cancellable**.

B2.3.14.1 CRYPTOKI UPDATE SHORTCUT

The "Cryptoki Update Shortcut" allows a service to streamline the processing of multi-stage cryptographic operations by installing an "update shortcut". Once installed, such a shortcut directly handles some commands coming from clients of the service and automatically calls the Cryptographic API on behalf of the service.

The typical use case of this feature is a DRM agent implemented as a secure native service within the Trusted Foundations. A player, using a DRM stub over the TF Client API, communicates with the DRM agent. The scenario is a decryption of an encrypted digital content, done by the DRM agent where the DRM agent securely manages the session key.

Without the shortcut, the typical sequence of operations is as follows:

1. The DRM Stub sends a command to the DRM service to setup a decryption session;
2. The DRM Agent opens the key for the decryption and starts a multi-part decryption operation by calling `C_DecryptInit`
3. The DRM Stub sends the encrypted data to the DRM Agent in multiple chunks, each chunk being sent in a separate command
4. When receiving such a command, the DRM Agent decrypts the data by calling `C_DecryptUpdate` and sends the result back to the DRM Stub
5. When the decryption is finished, the DRM Stub sends a final command to the DRM Agent
6. When the DRM Agent receives this command, it terminates the cryptographic operation by calling `C_DecryptFinal`

The "Cryptoki Update Shortcut" feature is specifically designed in order to optimize the implementation of this kind of use cases. The goal is to accelerate the cryptographic operations driven by the service.

With an update shortcut, the decryption path is optimized so that it no longer transits through the service:

1. The DRM Stub sends a command to the DRM service to setup a decryption session;
2. The DRM Agent opens the key for the decryption and starts a multi-part decryption operation by calling `C_DecryptInit`. **The agent then activates the shortcut using the `CV_ActivateUpdateShortcut2` function**
3. The DRM Stub sends the encrypted data to the DRM Agent in multiple chunks, each chunk being sent in a separate command
4. **The system intercepts the command and directly calls `C_DecryptUpdate` on behalf of the service. The service entry point `SRVXInvokeCommand` is not called**
5. When the decryption is finished, the DRM Stub sends a final command to the DRM Agent
6. When the DRM Agent receives this command, it terminates the cryptographic operation by calling `C_DecryptFinal`. **This implicitly deactivates the shortcut**.
7. **At any time, the service can also explicitly deactivate the shortcut using the `CV_DeactivateUpdateShortcut` function. The commands are then again routed to the service.**

Depending on the implementation, using a shortcut may significantly improve the performance and reactivity of the operation.

For an example of service that uses the Cryptoki Update Shorcut, see section A2.6, "*DRM Example*".

The Cryptoki Update Shortcut introduces two functions:

- `CV_ActivateUpdateShortcut2` creates a shortcut that connects a specific command identifier (value of the parameter `nCommandID` of the `SRVXInvokeCommand` entry point) in the current SRVX client session to a cryptoki operation in a given cryptoki session.
- `CV_DeactivateUpdateShortcut` destroys the shortcut.

Additionally, the shortcut modifies the behavior of the following functions:

- `C_EncryptFinal`, `C_DecryptFinal`, `C_DigestFinal`, `C_SignFinal`, and `C_VerifyFinal` (collectively referred as `C_`*`<Op>`*`Final`): these functions deactivate the shortcut registered in the cryptoki session, if any.
- `C_Encrypt`, `C_Decrypt`, `C_Digest`, `C_Sign`, and `C_Verify` (collectively referred to as `C_`*`<Op>`*): these functions deactivate the shortcut as well because they terminate the operation
- `C_CloseSession`: this function also deactivates the registered shortcut, if any.

Finally, the shortcuts activated within a given client session are all deactivated automatically when the service returns from the call to `SRVXCloseClientSession`.

B2.3.14.2 CV_ActivateUpdateShortcut2

```
CK_RV CV_ActivateUpdateShortcut2(
  CK_SESSION_HANDLE hCryptokiSession,
  uint32_t nCommandID,
  uint32_t nFlags,
  uint32_t nReserved
);
```

Installs a cryptoki update shortcut. Once activated successfully, a shortcut alters the processing of the commands when the client of the current SRVX session sends a command whose identifier is equal to `nCommandID` (the command identifier is the value of the parameter `nCommandID` passed to the function `TEEC_InvokeCommand`, `TEEC_InvokeCommandEx`, or `SXControlInvokeCommand`). In this case, the system intercepts the message and calls the cryptographic update function corresponding to the current cryptographic operation (`C_EncryptUpdate`, `C_DecryptUpdate`, etc.)

A service can call this function only if the following conditions are true:

- This function must be called from a session thread, otherwise a panic is raised.
- The handle `hCryptokiSession` must refer to a Cryptoki session (primary or secondary) that is opened and in which an operation has been initialized. This means that the service has started an operation by calling one of the `C_`*<Op>*`Init` functions (where *<Op>* stands for `Encrypt`, `Decrypt`, `Digest`, `Sign`, or `Verify`)
    - The operation mechanism must support multi-part processing (see section B4.2.5, "*Supported Mechanisms*" for a definition of what mechanism support muti-part operations), i.e., it must be possible to call `C_`*<Op>*`Update`. The service may already have called the function `C_`*<Op>*`Update` one or multiple times, but it must not have closed the operation using `C_`*<Op>*`Final` or `C_`*<Op>* yet
    - No shortcut must have been previously installed on the same `hCryptokiSession`
- No shortcut must have been previously installed in the same client session with the same `nCommandID`

When all these conditions are met, the system attempts to create a shortcut processing path. Once the shortcut is activated, the following processing takes place on each command sent by a client:

- First, the system determines if the command must take a shortcut path or the normal path. It must take the shortcut path if and only if:
    - There is a shortcut attached to the client session
    - The command identifier is equal to the value of the `nCommandID` passed to the function `CV_ActivateUpdateShortcut2`
- If the command does not take the shortcut path, it is delivered to the service as usual. Otherwise, the system parses the command parameters:
    - For all commands, the parameter #0 must be of type `MEMREF_INPUT` and must not be `NULL`. It contains the input data.
    - If the shortcut has been activated for an encryption or decryption operation, the parameter #1 must be of type `MEMREF_OUTPUT`. It contains a buffer for the output data.
    - Otherwise, for updates that generate no output, i.e., for digest, signature, or verification, then parameter #1 must be of type `NONE`.
    - Parameters #2 and #3 must be of type `NONE`.
    - If the parameters do not comply with these constraints, the system terminates the current cryptographic operation (and hence deactivates the shortcut) and directly answers with the error code `S_ERROR_BAD_PARAMETERS`.
- If the command is an encryption or decryption operation, the size of the parameter #1 is checked:
    - if the parameter is `NULL`, then the command is a request to retrieve the output length. In this case, the system calls the function `C_EncryptUpdate` or `C_DecryptUpdate` with a `NULL` output buffer, updates the size of the parameter and returns `S_SUCCESS`.

    - if the parameter is not `NULL` but its size is not large enough to contain the output data, then the system updates the size of the parameter and returns `CKR_BUFFER_TOO_SMALL`.
- The function `C_`*`<Op>`*`Update` is called
- If any error is returned by the function, the system deactivates the shortcut (because the cryptographic operation has actually terminated) and returns the error code `rv` to the client
- Otherwise, if the function returns `CKR_OK`, then:
    - for encryption or decryption operations, the system updates the size of parameter #1 with the actual output size

Note that a command handled by the shortcut may terminate the current cryptographic operation if the parameters are ill-typed or if the call to `C_`*`<Op>`*`Update` fails.

When a cryptoki update shortcut is activated for a given Cryptoki session, the service can still call cryptoki functions. Such a call can terminate the current cryptographic operation as described in the Cryptographic API Specification (see Chapter B4 "*Trusted Foundations* Cryptographic API"). In particular, this happens when the service calls the functions `C_`*`<Op>`*`Final` or `C_`*`<Op>`*.

When the cryptographic operation is terminated in this way, the shortcut is automatically deactivated.

This deactivation is atomic: if the client sends a command that can take the shortcut while the service is calling a function that terminates the current operation, then the behavior may only be one of the following:

- The client command arrives before the service calls the function. In this case, the command takes the shortcut and the Cryptoki session is guaranteed to be in the correct state to handle the call, i.e., the result will not fail because the operation has terminated.
- The client command arrives after the service calls the function. In this case, the command is routed to the service and the service must decide what to do with it. When the command arrives to the service, it is guaranteed that the cryptographic operation has terminated.

Notes

- This mechanism works for a normal-world client as well as for a secure-world client
- If the service and the client perform multiple updates in parallel, the order in which the updates occur is unpredictable, but it is guaranteed that the final result correspond to a particular ordering of the calls, i.e., no call is lost.
- There can be only one update shortcut activated for a given cryptoki session

Parameters

- `hCryptokiSession`: handle of the PKCS#11 session started within the secure service.
- `nCommandID`: Identifier of the command that needs to be used by a client of the service desiring to process data with the path established by the shortcut
- `nFlags`: Allows fine tuning of the shortcut implementation. The following flag is defined in this specification:
    - `S_UPDATE_SHORTCUT_FLAG_AGGRESSIVE`: the meaning of this flag is implementation-dependent. When supported, it usually selects an aggressively optimized shortcut implementation, which is likely to exhibit better performance that the default implementation, but may have other undesirable side effects. Consult the *Product Reference Manual* of your product for more details about this flag. Note that the default flags (`0`) always select the most conservative and safe shortcut implementation.
- `nReserved`: reserved for future use. Must be set to 0.

Return Value

- `CKR_OK`: indicates that the Cryptoki update shortcut has been activated with success.
- `CKR_OPERATION_ACTIVE`: there is already an installed shortcut on the same `hCryptokiSession` or there is already an installed shortcut in the same client session with the same `nCommandID`.

- `CKR_SESSION_HANDLE_INVALID`: the specified session handle was invalid at the time that the function was invoked.
- `CKR_OPERATION_NOT_INITIALIZED`: there is no active operation in the specified session or the operation does not support multi-part processing.
- `CKR_ARGUMENTS_BAD`: indicates that at least one of the values required for `nFlags` and `nReserved` is invalid.
- `CKR_HOST_MEMORY`: indicates the process in which the service is running has insufficient memory to perform the requested function.

Panics

- The function is not called from a session thread

B2.3.14.3 CV_DEACTIVATEUPDATESHORTCUT

```
void CV_DeactivateUpdateShortcut(
  CK_SESSION_HANDLE hCryptokiSession);
```

Deactivates a Cryptoki Update Shortcut.

After this function is called, the commands are again routed to the service for the client session and the command identifier of the shortcut.

This function does nothing if the handle `hCryptokiSession` is null or if no shortcut is currently activated in the session. It will panic if `hCryptokiSession` is an invalid cryptoki session handle.

Notes

- This function can be called at any time by the service, even in a non-session thread

Parameters

- `hCryptokiSession`: handle of the PKCS#11 session on which the update shortcut must be deactivated.

Panics

This function will panic if called with a cryptoki session that is not null and is invalid

B2.3.15 Service Control API

A secure service can communicate with another service using the service control functions specified in this section. In this case, the service acts as a client of the called service.

B2.3.15.1 SXControlOpenClientSession

```
S_RESULT SXControlOpenClientSession (
           const S_UUID*      pDestination,
           S_TIME_LIMIT*      pTimeLimit,
           uint32_t           nParamTypes,
           IN OUT S_PARAM     pParams[4],
           OUT S_HANDLE*      phSessionHandle,
           OUT uint32_t*      pnReturnOrigin)
```

This function opens a new session with a service.

The service is identified by its UUID.passed in `pidService`. This UUID can be hardcoded in the caller code or may be returned by querying the Service Manager.

The client may limit the time allowed for performing the connection by specifying an absolute time limit, constructed using `STimeGetLimit`, in the `pTimeLimit` parameter of this operation. The operation can also be explicitly cancelled by the client, which can be done by cancelling the thread that calls the function `SXControlOpenClientSession`. When the open operation times out or is cancelled the system will try to abort the operation. This request may or may not be honored by the called service. If it is acted upon, the function `SXControlOpenClientSession` will return the error code `S_ERROR_CANCEL` or any other error code determined by the called service.

An initial set of four parameters can be passed during the operation. See section B2.3.15.3 for a detailed specification of how these parameters are passed in the `nParamTypes` and `pParams` arguments.

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

- If the return origin is different from `S_ORIGIN_TRUSTED_APP`, then the function has failed before it could reach the target service. The possible error codes are listed below.
- If the return origin is `S_ORIGIN_TRUSTED_APP`, then the meaning of the return code depends on the protocol of the target service. However, if `S_SUCCESS` is returned, it always means that the session was successfully opened and if the function returns a code different from `S_SUCCESS`, it means that the session opening failed.

When the session is successfully opened, i.e., when the function returns `S_SUCCESS`, a valid session handle is written into `*phSessionHandle`. Otherwise, the value `S_HANDLE_NULL` is written into `*phSessionHandle`.

When a session is to be closed, the client service must call the function `SHandleClose` with the session handle.

Parameters

- `pDestination`: a pointer to a `S_UUID` structure containing the UUID of the destination service.
- `pTimeLimit`: The time limit after which the operation is aborted if it has not already been completed. Setting this value to `NULL` indicates an infinite timeout. If `pTimeLimit` is not `NULL`, then *`pTimeLimit` must have been created by calling `STimeGetLimit`.
- `nParamTypes`: the types of all parameters passed in the operation. See section B2.3.15.3 for more details.
- `pParams`: the parameters passed in the operation. See B2.3.15.3 for more details.

- `phSessionHandle`: a pointer to a variable that will receive the client session handle. The pointer must not be `NULL`. The value is set to `S_HANDLE_NULL` upon error.
- `pnReturnOrigin`: a pointer to a variable which will contain the return origin. This field may be `NULL` if the return origin is not needed.

Return Value

If the return origin is different from `S_ORIGIN_TRUSTED_APP`, one of the following error codes can be returned:

- `S_ERROR_OUT_OF_MEMORY`: not enough resources are available to open the session
- `S_ERROR_ITEM_NOT_FOUND`: if no service matches the requested destination UUID
- `S_ERROR_ACCESS_DENIED`: if access to the destination service is denied
- `S_ERROR_BUSY`: if the destination service does not allow more than one session
- `S_ERROR_CANCEL`: if the session opening operation has timed out or was cancelled before it reached the destination service
- `S_ERROR_TARGET_DEAD`: if the destination service has panicked during the operation

If the return origin is `S_ORIGIN_TRUSTED_APP`, the return code is defined by the protocol of the destination service. In any case, a return code set to `S_SUCCESS` means that the session was successfully opened and a return code different from `S_SUCCESS` means that the opening failed.

B2.3.15.2 SXControlInvokeCommand

```
S_RESULT SXControlInvokeCommand (
           S_HANDLE               hSessionHandle,
           const S_TIME_LIMIT*  pTimeLimit,
           uint32_t               nCommandID,
           uint32_t               nParamTypes,
           IN OUT S_PARAM        pParams[4],
           OUT uint32_t*         pnReturnOrigin)
```

This function invokes a command within a session opened between the caller's service and a destination service.

The parameter `hSessionHandle` must reference a valid session handle opened by `SXControlOpenClientSession`.

The client may limit the time allowed for performing the connection by specifying an absolute time limit, constructed using `STimeGetLimit`, in the `pTimeLimit` parameter of this operation. The operation can also be explicitly cancelled by the client, which can be done by cancelling the thread that calls the function `SXControlInvokeCommand`. When the open operation times out or is cancelled the system will try to abort the operation. This request may or may not be honored by the called service. If it is acted upon, the function `SXControlInvokeCommand` will return the error code `S_ERROR_CANCEL` or any other error code determined by the called service.

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

Up to four parameters can be passed during the operation. See section B2.3.15.3 for a detailed specification of how these parameters are passed in the `nParamTypes` and `pParams` arguments.

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

- If the return origin is different from `S_ORIGIN_TRUSTED_APP`, then the function has failed before it could reach the target service. The possible error codes are listed below.
- If the return origin is `S_ORIGIN_TRUSTED_APP`, then the meaning of the return code is determined by the protocol exposed by the destination service. It is recommended that the service developer chooses `S_SUCCESS` (0) to indicate success in their protocol, as this means that it is possible to determine success or failure without looking at the return origin.

Parameters

- `hSessionHandle`: a opened session handle.
- `pTimeLimit`: The time limit after which the operation is aborted if it has not already been completed. Setting this value to `NULL` indicates an infinite timeout. If `pTimeLimit` is not `NULL`, then *`pTimeLimit` must have been created by calling `STimeGetLimit`.
- `commandID`: the identifier of the Command within the Secure Service to invoke. The meaning of each Command Identifier must be defined in the protocol exposed by the Service.
- `nParamTypes`: the types of all parameters passed in the operation. See section B2.3.15.3 for more details.
- `pParams`: the parameters passed in the operation. See section B2.3.15.3 for more details.
- `pnReturnOrigin`: a pointer to a variable which will contain the return origin. This field may be `NULL` if the return origin is not needed.

Return Value

If the return origin is different from `S_ORIGIN_TRUSTED_APP`, one of the following error codes can be returned:

- `S_ERROR_OUT_OF_MEMORY`: not enough resources are available to perform the operation

- `S_ERROR_CANCEL`: if the operation has timed out or was cancelled before it reached the destination service
- `S_ERROR_TARGET_DEAD`: if the destination service has panicked

If the return origin is `S_ORIGIN_TRUSTED_APP`, the return code is defined by the protocol of the destination service.

B2.3.15.3 OPERATION PARAMETERS IN THE SXCONTROL API

The functions `SXControlOpenClientSession` and `SXControlInvokeCommand` take as arguments `nParamTypes` and `pParams`. The calling service can use these arguments to pass up to four parameters.

Each of the parameters has a type, which is one of the `S_PARAM_TYPE_XXX` constant. The content of `nParamTypes` should be built using the macro `S_PARAM_TYPES` (see section B2.3.4.6).

Unless all parameter types are set to `S_PARAM_TYPE_NONE`, `pParams` must not be `NULL` and must point to an array of four `S_PARAM` elements. Each of the `pParams[i]` is interpreted as follows.

When the operation starts, the system reads the parameters as follows:

Parameter Type	Interpretation
`S_PARAM_TYPE_NONE` `S_PARAM_TYPE_VALUE_OUTPUT`	Ignored
`S_PARAM_TYPE_VALUE_INPUT` `S_PARAM_TYPE_VALUE_INOUT`	Contains two integers in `pParams[i].value.a` and `pParams[i].value.b`.

`S_PARAM_TYPE_MEMREF_INPUT`
`S_PARAM_TYPE_MEMREF_OUTPUT`
`S_PARAM_TYPE_MEMREF_INOUT`
	`pParams[i].memref.pBuffer` and `pParams[i].memref.nSize` must be initialized with a memory buffer that is accessible with the access rights described in the type. The buffer can be `NULL`, in which case `nSize` must be set to 0

During the operation, the destination service can update the contents of the `OUTPUT` or `INOUT` memory references.

When the operation completes, the system updates the structure `pParams[i]` as follows:

Parameter Type	Interpretation

`S_PARAM_TYPE_NONE`
`S_PARAM_TYPE_VALUE_INPUT`
`S_PARAM_TYPE_MEMREF_INPUT`
	Unchanged

`S_PARAM_TYPE_VALUE_OUTPUT` `S_PARAM_TYPE_VALUE_INOUT`	`pParams[i].value.a` and `pParams[i].value.b` are updated with the value sent by the destination service
`S_PARAM_TYPE_MEMREF_OUTPUT` `S_PARAM_TYPE_MEMREF_INOUT`	`pParams[i].memref.nSize` is updated to reflect the actual or requested size of the buffer

B2.3.16 Miscellaneous Generic Functions

This section contains the definition of the following generic functions:

- `SPanic` can be used to raise a panic when a service detects a critical error. A critical error can be a critical security error, an error that prevents correct functioning of the service, or the detection that a precondition is not met.
- `SHandleClose` is a generic function to close all types of handles

B2.3.16.1 SPanic

```
void SPanic(S_RESULT nPanicCode);
```

The `SPanic` function raises a service panic.

When a service calls the `SPanic` function the System kills the service process; all the pending service functions are terminated, the instance is destroyed, and all the resources opened by the service are reclaimed. No service function of the instance is ever called again after a panic.

When a service panics, the clients of this service receive the error code `S_ERROR_TARGET_DEAD` or origin `S_ORIGIN_TEE` until they close their session with the service. This applies to normal world clients calling through TF Client API or to secure world clients calling through the `SXControl` API.

The `SPanic` function does not return to the calling thread; the panic will occur within the execution of this function.

Once an instance is panicked, no service entry point is ever called again for this instance, not even `SRVXDestroy`.

Parameters

- `nPanicCode`: a panic code. Will be displayed in traces if traces are available.

B2.3.16.2 SHandleClose

```
void SHandleClose( S_HANDLE hHandle )
```

The `SHandleClose` function is a generic function that can close all the types of handles. The following handles may be closed:

- Handles on threads,
- Handles on semaphores,
- Handles on files,
- Handles on file enumerations,
- Handles on cryptographic sessions,
- Handles on cryptographic objects,
- Handles on `SXControl` client session.

The effect of the function depends on the type of the object denoted by the handle:

- If `hHandle` denotes a handle on a thread, closing the handle makes it unusable. If the thread is active, it is not stopped. The thread object is released when the thread entry point function has returned and when the thread handle has been closed.
- If `hHandle` denotes a handle on a semaphore, closing the handle makes it unusable. If one or several threads are currently waiting on the semaphore, these threads will be woken up only if they are cancelled or the time out expires.
- If `hHandle` denotes a handle on a file, closing the handle frees the resources associated with the file handle. The file object itself is released when all the handles opened on the file are closed and when all the operations on the file have completed.
- If `hHandle` denotes a handle on a file enumeration, closing the handle frees the resources associated with the file enumeration.
- If `hHandle` denotes a handle on a cryptographic session, then this function is equivalent to `C_CloseSession`.
- If `hHandle` denotes a handle on a cryptographic object, then the reference count on the object handle is decremented. When the reference count reaches zero, the handle is no longer usable. Note that an object handle can be returned multiple times by the Cryptographic API. Note also that whenever a cryptographic session is closed, all the object handles are implicitly closed as well.
- If `hHandle` denotes a handle on a client session opened using the function `SXControlOpenClientSession`, then this function closes the session. This first aborts any command pending in the session.

Note that if the value of `hHandle` is `S_HANDLE_NULL`, this function does nothing and simply returns.

Whether the function `SHandleClose` is preemptable or not depends on the type of the object denoted by the handle:

- Closing a file, a file enumeration, an object-store, a cryptographic session, a cryptographic object or a client session is **preemptable**.
- Closing other types of handles is **not preemptable**.

The function `SHandleClose` is never cancellable.

Parameters

- `hHandle`: The handle to close.

Panic Reasons

- The handle `hHandle` value is different from `S_HANDLE_NULL` and does not reference a live handle of one of the handle types described above

B2.3.17 Implementation Functions

These functions can be called from any thread and return information about the implementation.

B2.3.17.1 SImplementationGetAllProperties

```
S_RESULT SImplementationGetAllProperties(
       OUT S_PROPERTY** ppProperties,
       OUT uint32_t*    pnPropertiesCount )
```

Returns the list of all properties associated with the implementation.

The implementation allocates a single memory block containing an array of `S_PROPERTY` structures and the corresponding strings. Each element of the array contains a pointer to a single property's name and value. The name and value are zero-terminated Unicode strings encoded in UTF-8. These strings are part of the memory block allocated by the implementation.

Upon successful completion, the calling service must call the `SMemFree` function to deallocate the properties block. Calling `SMemFree` will free the property array as well as each property’s name and value strings.

This function is **not preemptable**.

The following implementation properties are defined:

Table B2-11: SSDI: Implementation Properties

Property Name	Type	Meaning
`ssdi.apiversion`	String	The version number of the SSDI. Its value is either “3.0” or “3.1” depending on the API version actually implemented by your product..
`ssdi.apidescription`	String	A description of the implementation. The content of this property is implementation-dependent but typically contains a version and build number of the implementation as well as other configuration information.
`date.protectionLevel`	Integer	The protection level provided by the date implementation. See the function `SDateGet` for more details.

Note that implementation properties are fixed and services can assume they never change during the life-time of a device.

Parameters

- `ppProperties`: A pointer to a variable that will receive a pointer to the array of properties. This variable is set to `NULL` upon error
- `pnPropertiesCount`: A pointer to a variable that will receive the number of elements in the returned array. This variable is set to zero upon error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.

Panic Reasons

This function may raise a panic for the following reasons:

- `ppProperties` or `pnPropertiesCount` is `NULL`.

B2.3.17.2 SImplementationGetProperty

```
S_RESULT SImplementationGetProperty(
        IN const char* pName,
       OUT char**      ppValue )
```

The `SImplementationGetProperty` function retrieves a single implementation property. See `SImplementationGetAllProperties` for a definition of the implementation properties supported.

The implementation allocates the string returned. The caller must deallocate this string with `SMemFree`.

This function is **not preemptable**.

Parameters

- `pName`: The name of the property to retrieve.
- `ppValue`: The value of the property, or `NULL` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_OUT_OF_MEMORY` if there is not enough memory to complete the operation.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.

Panic Reasons

This function may raise a panic for the following reasons:

- `pName` is `NULL`.
- `ppValue` is `NULL`.

B2.3.17.3 SImplementationGetPropertyAsInt

```
S_RESULT SImplementationGetPropertyAsInt (
         IN const char* pName,
        OUT uint32_t*   pnValue )
```

The `SImplementationGetPropertyAsInt` function retrieves a single implementation property and converts it to an integer. See `SImplementationGetAllProperties` for a definition of the implementation properties supported.

Refer to section B2.3.2.1 for details on the valid integer formats for the property file.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the name of the property to retrieve.
- `pnValue`: A pointer to the variable that will contain the value of the property on success, or zero on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.
- `S_ERROR_BAD_FORMAT` if the property value cannot be converted to a positive integer.

Panic Reasons

This function may raise a panic for the following reasons:

- `pName` is `NULL`.
- `pnValue` is `NULL`.

B2.3.17.4 SImplementationGetPropertyAsBool

```
S_RESULT SServiceGetPropertyAsBool(
        IN const char* pName,
       OUT bool*       pbValue )
```

The `SImplementationGetPropertyAsBool` function retrieves a single implementation property and converts it to a Boolean.

Refer to section B2.3.2.2 for details on the valid Boolean formats for the property file.

See `SImplementationGetAllProperties` for a definition of the implementation properties supported.

This function is **not preemptable**.

Parameters

- `pName`: A pointer to the name of the property to retrieve.
- `pbValue`: A pointer to the variable that will contain the value of the property on success, or `false` on error.

Return Value

- `S_SUCCESS` in case of success.
- `S_ERROR_ITEM_NOT_FOUND` if the property is not found.
- `S_ERROR_BAD_FORMAT` if the property cannot be converted to a boolean

Panic Reasons

This function may raise a panic for the following reasons:

- `pName` is `NULL`.
- `pbValue` is `NULL`.

Chapter B3 TRUSTED FOUNDATIONS CLIENT API

This chapter is the specification of the Trusted Foundations Client API, which is the API available to Normal World clients to access secure services.

B3.1 ABOUT THE TRUSTED FOUNDATIONS CLIENT API

The Trusted Foundations Client API (or "TF Client API") defines the external interface in the C language to the Trusted Foundations for communicating with secure services. It is based on Global Platform's "TEE Client API" V1.0 with a few extensions.

This chapter specifies the TF Client API in the following sections:

- Section B3.2, "*Introduction to the TF Client API*" provides an introduction to the features of the TF Client API;
- Section B3.3, "*Specification of the TF Client API*" provides a complete specification of the TF Client API.

B3.2 INTRODUCTION TO THE TF CLIENT API

The TF Client API defines a communications API for connecting *Client Applications* running in a rich operating environment with the *Secure Services* running inside the Trusted Foundations execution environment.

B3.2.1 DESIGN PRINCIPLES

The TF Client API is based on Global Platform's TEE Client API whose main design principles are recalled here:

- **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.
- **Client-side memory allocations**:
  - Where possible the design of the TF 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 TF Client API are chosen to maximize the possibility of zero-copy data transfer between the Client Application and the Secure Service, 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 TF Client API is used to implement higher-level APIs, such as cryptography or secure storage, where the caller often provides memory buffers for input or output data using simple C pointers. The TF Client API allows 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 TF 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 Secure Services. These are defined by other specifications

B3.2.2 FUNDAMENTAL CONCEPTS

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

B3.2.2.1 TEE CONTEXTS

A *TEE Context* is an abstraction of the logical connection which exists between a Client Application and the Trusted Foundations. A TEE Context must be initialized before a Session can be created between the Client Application and a Secure Service running within the Trusted Foundations represented by that TEE Context. 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 instance of the Trusted Foundations, or with multiple instances of Trusted Foundations if they are available. The number of concurrent contexts which may exist is *implementation-defined*, and may additionally depend on run-time resource constraints.

B3.2.2.2 SESSIONS

A *Session* is an abstraction of the logical connection which exists between a Client Application and a specific Secure Service. 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 Secure Services in use, and may additionally depend on run-time resource constraints.

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

Connection Methods: Login

Some Secure Services may require the Implementation to identify or authenticate the Client Application or the user executing it. For example, a Secure Service 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 Secure Service in use; attempting to open a Session with an incorrect connection method may result in a failed attempt.

B3.2.2.3 COMMANDS

A *Command* is the unit of communication between a Client Application and a Secure Service within a Session. When starting a new Command the Client Application identifies the function in the Secure Service which it wishes to execute by passing a numeric command identifier, and may also provide an operation payload in accordance with the protocol the Secure Service exposes for that function. The Command invocation blocks the Client Application thread, waiting for an answer from the Secure Service. 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 Secure Services 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* 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 ("inout"). For Memory Reference Parameters, the specified direction of data flow determines when the underlying memory buffers need to be synchronized with the Secure Service.

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 Secure Service function in use, and hence outside of the scope of this specification.

Temporary Memory References

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 B3.2.2.4) 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 make use of the common error codes listed in section B1.2 and the return origins are defined in section B3.3.4.2.

When the return origin is `TEEC_ORIGIN_TRUSTED_APP` then the return code is defined by the Secure Service's protocol. Note that, critically, this means that a Client Application cannot just test against `TEEC_SUCCESS`, as the Secure Service may use another code to indicate success. To enable simpler error handling code in the Client Application it is recommended that the Secure Service 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 Secure Service 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 Secure Service until the event occurs inside the Trusted Foundations. When the event occurs the Secure Service passes control back to the Client Application; the `TEEC_InvokeCommand` will return and the Client Application can handle the event which was signaled.

B3.2.2.4 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 Secure Service.

A Shared Memory block can either be existing Client Application memory which is subsequently registered with the TF Client API, or memory which is allocated on behalf of the Client Application using the TF 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.

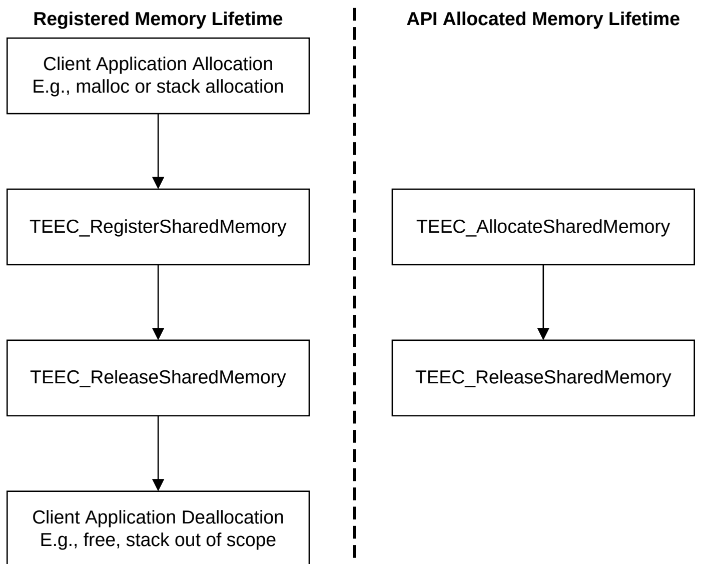

Registered Memory Lifetime
Client Application Allocation
E.g., malloc or stack allocation
TEEC_RegisterSharedMemory
TEEC_ReleaseSharedMemory
Client Application Deallocation
E.g., free, stack out of scope
API Allocated Memory Lifetime
TEEC_AllocateSharedMemory
TEEC_ReleaseSharedMemory

Figure B3-1: Shared Memory Buffer Lifetime.

Zero-copy Data Transfer

When possible the implementation of the communications channel beneath the TF Client API tries to directly map Shared Memory in to the Secure Service memory space, enabling true zero-copy data transfer. However this is not always possible; for example, the Trusted Foundations may run on a processor that does 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 TF Client API Implementation is allowed to synchronize the data in a Shared Memory block with the Trusted Foundations to ensure data consistency. The Client Application and Secure Service 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 B3.2.2.5.

Client Application developers should note that letting the TF 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 Secure Service 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 B3.2.2.5.

B3.2.2.5 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_RegisteredMemoryReference` structure 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 Secure Service), output (buffer is transferring data from the Secure Service 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 B3-2: Memory Reference timing diagram:

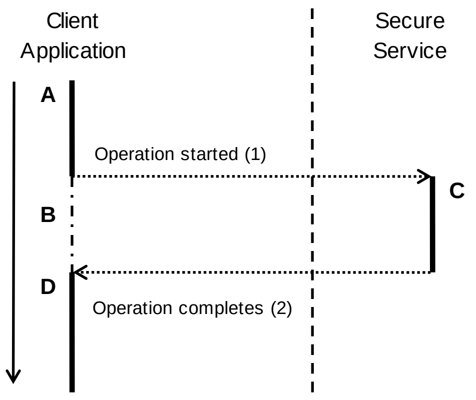


Figure B3-2: Memory Reference timing diagram

In this figure there are three temporal states for the Client application (A, B, D) and one for the Secure Service (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 Secure Service'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 Secure Service'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 Secure Service 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 Secure Service 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 Secure Service may read and write to the memory range (state C).

If these synchronization rules are ignored by the Client Application or the Secure Service 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 Secure Service 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 Secure Service's protocol. A Secure Service 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 Secure Services 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 Secure Service 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 Secure Service.

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

B3.2.2.6 VARIABLE LENGTH RETURN BUFFERS

In many cases the Secure Service 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 is the maximum size of the output data that the Secure Service may write into the referenced region. When the Secure Service 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 Secure Service.

In these cases the Implementation only needs to synchronize the number of bytes which the Secure Service 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 Secure Service 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 Secure Service 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 Secure Service 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.

B3.2.3 Usage Concepts

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

B3.2.3.1 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.

B3.2.3.2 Multi-threading

The TF 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 Secure Service 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.

B3.2.3.3 Resource Cleanup

The specification of the "stop" functions described in section B3.2.3.2 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.

B3.2.4 Security Aspects

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

B3.2.4.1 Security of the Trusted Foundations and Trusted Applications

The implementation of any Secure Services must treat any input from the rich environment as potentially malicious; Client Applications are running outside of the Trusted Foundations 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 Secure Service developer:

- Shared Memory is memory owned by the rich environment and mapped into the Secure Service memory space. Code inside the Secure Services 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 B3.2.2.5, 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 Secure Service. This identity information is generated based on parameters controlled by some trusted entity inside the rich operating system, such as the OS kernel. Secure Service developers must therefore note that the validity of this login token is bounded by the security of the rich operating system, not the security of the Trusted Foundations.

B3.2.4.2 Security of the Rich Operating System

In most implementations, the Trusted Foundations is a separate operating system which exists in parallel to the rich operating system which runs the Client Applications. The implementation always ensures that Client Applications cannot use the features exposed by the TF Client API to bypass the security sandbox used by the rich operating system to isolate processes.

B3.2.5 Non-Standard Extensions

The TF Client API introduces a few extensions to the Global Platform standard TEE Client API specification:

- The functions `TEEC_OpenSessionEx` and `TEEC_InvokeCommandEx` extend `TEEC_OpenSession` and `TEEC_InvokeCommand` with a time limit after which the operation is automatically cancelled;
- The functions `TEEC_GetImplementationInfo` and `TEEC_GetImplementationLimits` can be used to query information about the underlying implementation
- The login type `TEEC_LOGIN_PRIVILEGED` is introduced: it filters Client Applications depending on an OS-specific mechanism;
- The login type `TEEC_LOGIN_AUTHENTICATION` for Client Applications to request authentication through a cryptographic signature. This is described in more details in the next section.

B3.2.6 CLIENT AUTHENTICATION

The TF Client API includes a non-standard "Client Authentication" mechanism that can be used to control the access to services.

This mechanism is useful when a secure service might send sensitive information to its client. For example, a DRM agent may implement all the license management and content decryption but leave the decoding work to a codec located in its client application. In this case, it would be important to guarantee the authenticity of the client application.

The authentication process is triggered when an application opens a session using the login type `TEEC_LOGIN_AUTHENTICATION` (see the function TEEC_OpenSession).

There is no definite way for the client to know it must use authentication. This must be described in a specific client-service protocol. Some services may give unauthenticated clients access to some restricted functionalities and open more features to authenticated clients. Some other services may always require the client to be authenticated. Some services may also advertise their requirements in their properties so that the clients can adapt.

B3.2.6.1 SIGNATURE FILE

In order to guarantee the authenticity of the application, a signature file must be provided. This signature file contains the authenticity proof of the application and securely binds a set of properties with the application.

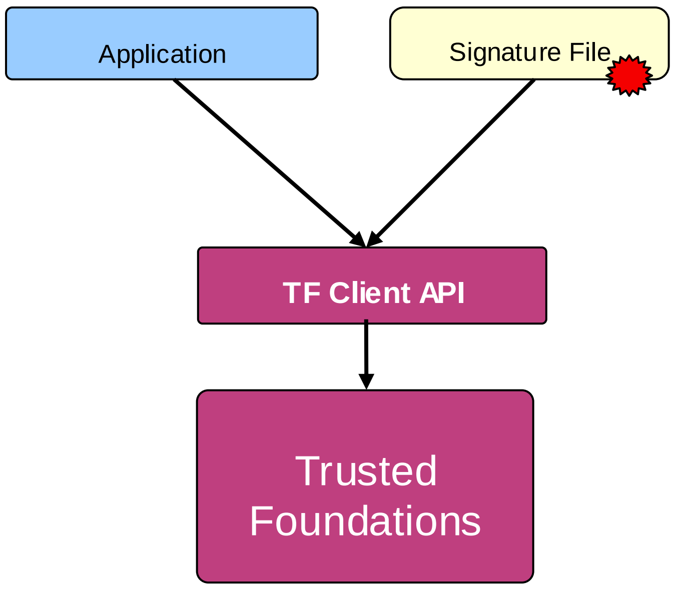


Figure B3-3: TF Client API: using Authenticated Login

A signature file contains the following elements:

- A manifest file associated with the application: the manifest file contains mandatory and optional properties, as defined in Section B3.3.6.2, "*Manifest File Format*". The manifest file contains:
    - a mandatory application UUID, which must uniquely identify the application. The system will not distinguish two authenticated applications with the same UUID;
    - Client authentication data, which must characterize the application in a persistent and tamper-proof manner. For example, the client authentication data can be a hash of the client application binary or can leverage an existing secure application identifier provided by the Operating System;
    - Other client properties, which can be used to enforce a service-specific acces policy.
- A signature of the manifest file. The signing key is a RSA private key
- Optionally, a chain of certificates that authenticates the signing key

The signature file is generated using a signature process similar to the one shown in the following figure:

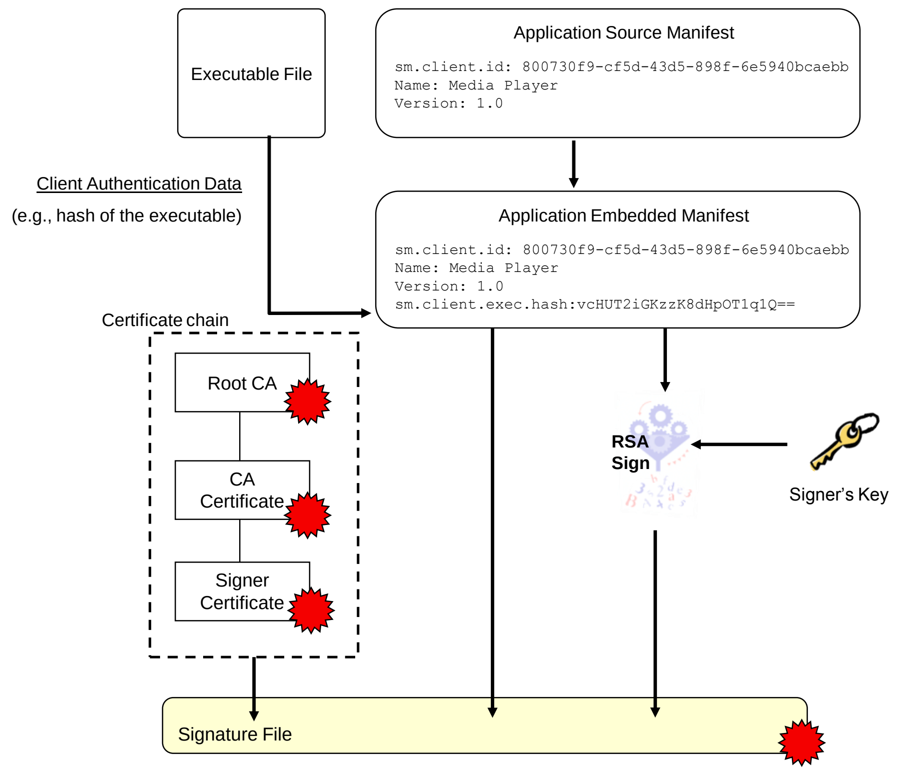

Executable File
Application Source Manifest
sm.client.id: 800730f9-cf5d-43d5-898f-6e5940bcaebb
Name: Media Player
Version: 1.0
Client Authentication Data
(e.g., hash of the executable)
Application Embedded Manifest
sm.client.id: 800730f9-cf5d-43d5-898f-6e5940bcaebb
Name: Media Player
Version: 1.0
sm.client.exec.hash:vcHUT2iGKzzK8dHpOT1q1Q==
Certificate chain
Root CA
CA Certificate
Signer Certificate
RSA Sign
Signer's Key
Signature File

Figure B3-4: TF Client API: Typical Signature File Generation

B3.2.6.2 Root Authority and the Certificate Chain

The Trusted Foundations contains an RSA public key of the root authority; this key is implicitly trusted and provides the basis of the security for the authentication process. The integrator must ensure that this public key is protected in integrity. If an attacker could change this root key, he could masquerade as an authenticated client application, and potentially gain access to privileged functionality or secret data of the client that they have stolen the identify of.

This root authority key can be used in two ways; either to directly verify the signature in the signature file, or to validate the first certificate in a certificate chain that contains the signer certificate that was used to sign the manifest file. This Public Key Infrastructure is based on X509 certificates, and distinguishes two kinds of certificate:

- *Signer Certificates* that belong to the entities that sign the manifest files.
- *Certificate Authority (CA) Certificates* that are used to establish a trust relationship between the signer certificates and the root authority.

By incorporating CA certificates into the authentication protocol, the scheme allows root authorities to establish trust relationships with other entities in order to delegate the signing of application manifest file. When trust relationships are used to generate a Signature file, then the certificate chain must be included in the Signature file.

- The root authority can be a CA; allowing it to sign, and hence develop a trust relationship with, CA certificates and signer certificates.
- The root authority can be a signer; allowing it to sign manifest files.
- The root authority can be a CA *and* a signer.

A simple example of a root authority that establishes trust relationship in order to delegate the signing of manifest files is shown in the following figure:

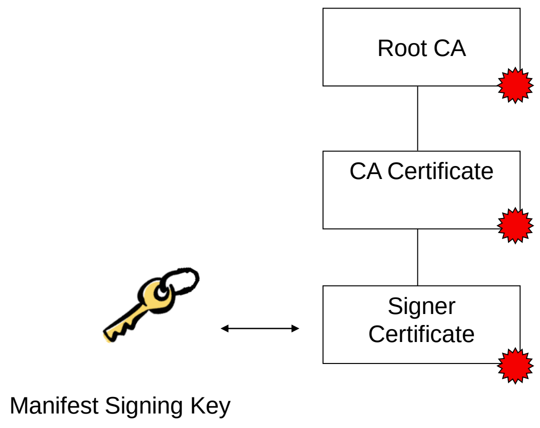


Figure B3-5: TF Client API: Example of Client Authentication Certificate Chain

The root authority key is used as a CA to sign a second CA certificate. This demonstrates that the root authority trusts the entity that controls the CA certificate it is signing.

The second CA certificate is used to sign a signer certificate. This demonstrates that the second CA trusts the entity that controls the signer certificate it is signing.

B3.3 Specification of the TF Client API

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

B3.3.1 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 the actual behavior is not specified and may depend on the actual product you are using.

Programmer Error

A *programmer error* can only be triggered by incorrect use of the API 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 is implementation-defined and could include a failing assertion, an informative return code if the function can return one, a diagnostic log file, etc. In any case, the implementation still guarantees the stability and security of the Trusted Foundations 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.

B3.3.2 Header File

The header file for the TF Client API has the name “`tee_client_api.h`”.

```
#include “tee_client_api.h”
```

B3.3.3 Data Types

B3.3.3.1 Basic Types, TEEC_UUID, TEEC_RESULT

The TF Client API makes use of the common types defined in section B1.1, “*Common Types*”.

B3.3.3.2 TEEC_Context

This type denotes a TEE Context, the main logical container linking a Client Application with the Trusted Foundations. Its content is entirely *implementation-defined*, which means that the client must not attempt to access the `imp` field.

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

B3.3.3.3 TEEC_Session

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

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

B3.3.3.4 TEEC_SharedMemory

This type denotes a Shared Memory block which has either been registered with the API 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 Trusted Foundations
- `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 Trusted Foundations
    - `TEEC_MEM_OUTPUT`: the memory can be used to transfer data from the Trusted Foundations 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

B3.3.3.5 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`, this field is updated to reflect the actual or required size of the output:
    - If the destination secure service has actually written some data in the output buffer, then the `size` field is updated 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 `size` field is updated with the size of the output buffer requested by the secure service. In this case, the client must not assume that any data has been written into the output buffer. See the section B3.2.2.6, "*Variable Length Return Buffers*" for more details.

B3.3.3.6 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 only interprets 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”, this field is updated to reflect the actual or required size of the output. This applies even if the parameter type is `TEEC_MEMREF_WHOLE`:
        - If the secure service has actually written some data in the output buffer, then the `size` field is updated with the actual number of bytes written.
        - If the output buffer was not large enough to contain the whole output, the `size` field is updated with the size of the output buffer requested by the Secure Service. In this case, the Client Application must assume that no data has been written into the output buffer. See the section B3.2.2.6, “*Variable Length Return Buffers*” for more details.
- `offset` is the offset, in bytes, of the referenced memory region from the start of the Shared Memory block:
    - The Implementation only interprets this field if the Memory Reference type in the operation structure is not `TEEC_MEMREF_WHOLE`. Otherwise, the Implementation uses the base address of the Shared Memory block.

B3.3.3.7 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 Secure Service to assign a semantic to these two integers.

B3.3.3.8 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`

`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`

B3.3.3.9 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 should be considered implementation-defined and you must use the macro `TEEC_PARAMS_TYPE`, defined in section B3.3.5.10, to construct a constant value for this field. As a special case, if the Client Application sets `paramTypes` to 0, then the Implementation interprets 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 B3-3 in section B3.3.4.3:
    - `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 B3.3.3.8).
- `imp` contains any additional *implementation-defined* data attached to the operation structure.

B3.3.3.10 TEEC_TimeLimit

```
typedef struct
{
   uint32_t  x;
   uint32_t  y;
} TEEC_TimeLimit;
```

The time limit structure is an opaque type that holds the time since an arbitrary time origin. It is used by the functions `TEEC_OpenSessionEx` and `TEEC_InvokeCommandEx` to set a timeout on an operation to be performed. The content of the fields `x` and `y` is implementation-specific and must not be interpreted or modified by the client.

Each TEE context has an associated time origin, which remains unchanged until the device context is finalized. No other guarantees about the origin are made.

All instances of this structure must be created through the use of the function `TEEC_GetTimeLimit`; otherwise the implementation behavior is undefined. For this reason all time limits are relative to the time that the function `TEEC_GetTimeLimit` was called.

The use of time limits in this API should only be considered as hints to the underlying implementation; both the underlying system implementation or service may chose to ignore the time limit in some circumstances.

If an object of type `TEEC_TimeLimit` is used in a device context other than the device context in which it was created, the behavior is undefined.

B3.3.3.11 TEEC_ImplementationInfo

```
typedef struct
{
   char apiDescription[65];
   char commsDescription[65];
   char TEEDescription[65];
}
TEEC_ImplementationInfo;
```

This structure is used in the function TEEC_GetImplementationInfo documented in section B3.3.5.14. It contains strings describing various part of the implementation.

B3.3.3.12 TEEC_ImplementationLimits

```
typedef struct
{
   uint32_t pageSize;
   uint32_t tmprefMaxSize;
   uint32_t sharedMemMaxSize;
   uint32_t nReserved3;
   uint32_t nReserved4;
   uint32_t nReserved5;
   uint32_t nReserved6;
   uint32_t nReserved7;
}
TEEC_ImplementationLimits;
```

This structure is used in the function TEEC_GetImplementationLimits documented in section B3.3.5.15. It contains various implementation limits that a Client might be interested in.

B3.3.4 Constants

The following constants are defined by this specification:

B3.3.4.1 Return Codes

The TF Client API makes use of the common error codes defined in section B1.2.

B3.3.4.2 Return Code Origins

The following function return code origins, of type `uint32_t`, are defined by the TF Client API. 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`	`0x00000001`	The return code is an error that originated within the TF Client API implementation.
`TEEC_ORIGIN_COMMS`	`0x00000002`	The return code is an error that originated within the underlying communications stack linking the rich OS with the Trusted Foundations.
`TEEC_ORIGIN_TEE`	`0x00000003`	The return code is an error that originated within the Trusted Foundations implementation.
`TEEC_ORIGIN_TRUSTED_APP`	`0x00000004`	The return code originated within the Secure Service code. This includes the case where the return code is a success.
All other values Reserved for Future Use		

Table B3-1: TF Client API Return Code Origin Constants

B3.3.4.3 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 Secure Service.
`TEEC_MEM_OUTPUT`	`0x00000002`	The Shared Memory can carry data from the Trusted Application to the Secure Service.
All other flag values Reserved for Future Use		

Table B3-2: TF Client API Shared Memory Control Flags

B3.3.4.4 Parameter Types

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

Name	Value	Comment
`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 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.
`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 B3-3: TF Client API Parameter Types

B3.3.4.5 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 Secure Service.

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` depends on the identity of the application program, and not the user running it. If two users use the same program, the Implementation assigns the same login identity to both users so that they can access the same assets held inside the Trusted Foundations. These identity tokens are also persistent across multiple invocations of the application and across power cycles, enabling them to be used to disambiguate persistent storage.

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.
`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.
`TEEC_LOGIN_PRIVILEGED`	`0x80000002`	Indicates that the client application wishes to use the privileged login method to connect to a. The OS will perform an implementation-specific access control.
`TEEC_LOGIN_AUTHENTICATION`	`0x80000000`	Indicates that the client application wishes to use the authentication login method to connect to a service. A signature file must be provided when opening a session. See section B3.3.5.6, “TEEC_OpenSession” for more details.

Table B3-4: TF Client API Session Login Methods

B3.3.5 Functions

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

B3.3.5.1 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 Trusted Foundations identified by the string identifier `name`.

The Client Application may pass a `NULL` `name`. In this case, the implementation selects a default instance of the Trusted Foundations to connect to. The supported name strings, the mapping of these names to a specific instance of Trusted Foundations, and the nature of the default Trusted Foundations instance are *implementation-defined*.

The caller must pass a pointer to a valid TEE Context in `context`. This structure does not need to be initialized because the implementation assumes that all fields of the `TEEC_Context` structure are in an *undefined* state when the function is called.

Parameters

- `name`: a zero-terminated string that describes the Trusted Foundations instance to connect to. If this parameter is set to `NULL` the Implementation selects a default instance.
- `context`: a `TEEC_Context` structure that is initialized by the Implementation.

Return

- `TEEC_SUCCESS`: the initialization was successful.
- `TEEC_ERROR_OUT_OF_MEMORY`: not enough resource is available to initialize the TEE Context
- `TEEC_ERROR_COMMUNICATION`: the Trusted Foundations is not accessible or is in a "system panic" state

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 does not occur.

B3.3.5.2 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 Trusted Foundations. 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 cannot fail: after this function returns, the Client Application can always consider that the Context has been closed.

The function implementation does 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`.

B3.3.5.3 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 Trusted Foundations. It must be a combination of one or two of the following flags:
    - `TEEC_MEM_INPUT`: for data transfers from the Client Application to the Trusted Foundations;
    - `TEEC_MEM_OUTPUT`: for data transfers from the Trusted Foundations to the Client Application.
- The Implementation assumes that all other fields in the Shared Memory structure have *undefined* content.

An Implementation may put a limit on the size of a single Shared Memory block. These limits can be retrieved by using the function `TEEC_GetImplementationLimits` defined in section B3.3.5.15. 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.

Once successfully registered, the Shared Memory block can be used for efficient data transfers between the Client Application and the Secure Service. The implementation will attempt to transfer data without using copies. However, in some implementation or in some cases, the implementation will 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.

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.
- `TEEC_ERROR_COMMUNICATION`: the Trusted Foundations is unreachable or has entered an unrecoverable "system panic" state.

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.

B3.3.5.4 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, a non-`NULL` pointer is written in to the `buffer` field. It must never be dereferenced by the Client Application but can always be used in Registered Memory References.
- The `flags` field indicates the allowed directions of data flow between the Client Application and the Trusted Foundations. . It must be a combination of one or two of the following flags:
    - `TEEC_MEM_INPUT`: for data transfers from the Client Application to the Trusted Foundations;
    - `TEEC_MEM_OUTPUT`: for data transfers from the Trusted Foundations to the Client Application.
- The Implementation assumes that all other fields in the Shared Memory structure have *undefined* content.

An Implementation may put a limit on the size of a single Shared Memory block. These limits can be retrieved by using the function `TEEC_GetImplementationLimits` defined in section B3.3.5.15. 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.

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

Once successfully allocated the Shared Memory block can be used for efficient data transfers between the Client Application and the Secure Service. The implementation will attempt to transfer data in to the TEE without using copies, but may have to fall back on data copies if zero-copy cannot be achieved.

The memory buffer allocated by this function is guaranteed to 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.

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 in accordance with the specification described above.
    - On return, for a successful allocation the Implementation sets the pointer `buffer` to the address of the allocated block, otherwise it sets `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.

- `TEEC_ERROR_COMMUNICATION`: the Trusted Foundations is unreachable or has entered an unrecoverable “system panic” state.

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.

B3.3.5.5 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 frees the underlying memory and the Client Application must not access this region after this function has been called. In this case the Implementation sets the `buffer` and `size` fields of the `sharedMem` structure to `NULL` and `0` respectively before returning.

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

The Implementation does 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.

B3.3.5.6 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 Secure Service.

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

The target Secure Service 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 Secure Service to implement access control methods which separate functionality or data accesses for different actors in the rich environment outside of the Trusted Foundations. Additional data associated with each connection method is passed in via the pointer `connectionData` and depends on the login type:

- `TEEC_LOGIN_PUBLIC`, `TEEC_LOGIN_USER`, `TEEC_LOGIN_APPLICATION`, `TEEC_LOGIN_USER_APPLICATION`, `TEEC_LOGIN_PRIVILEGED`:
    - `connectionData` should be `NULL` and is ignored by the implementation
- `TEEC_LOGIN_GROUP`, `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.
    - `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_AUTHENTICATION`:
    - `connectionData` should be `NULL` and is ignored by the implementation. The signature file must be passed in parameter #3 as an input memory reference, which can be a temporary memory reference or a registered memory reference. The content of the memory reference must not be `NULL` and must comply with the file format defined in section B3.3.6, "*Signature File Format Specification*".

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 Secure Service requires a static login credential then this can be passed by the Client Application using one of the four Parameters in the Operation Payload.

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 parameters are exchanged with the Secure Service, and the operation cannot be cancelled by the Client Application. The full behavior of the Operation Payload handling is described in section B3.3.5.8, 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 is one of standard error codes and the error conditions are described below.

- 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

In any case, 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.

As a consequence, note that Secure Services 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 Secure Service.

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 Secure Service.
- `connectionMethod`: the method of connection to use. Refer to section B3.3.4.5 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`, one of the following error codes:
  - `TEEC_ERROR_BAD_PARAMETERS`:

    the login type is not one of the supported types or, in the case of the login `TEEC_LOGIN_AUTHENTICATION`, the parameter #3 is not an input memory reference or is `NULL`
  - `TEEC_ERROR_OUT_OF_MEMORY`:

    not enough resource is available to open the session
  - `TEEC_ERROR_ITEM_NOT_FOUND`:

    A destination service with the specified UUID was not found
  - `TEEC_ERROR_ACCESS_DENIED`:

    if the login procedure fails. This happens in the following conditions:
    - `TEEC_LOGIN_PRIVILEGED`: the OS rejected the access of the Client Application as “privileged”
    - `TEEC_LOGIN_GROUP` or `TEEC_LOGIN_GROUP_APPLICATION`: the Client Application does not belong to the requested group
    - `TEEC_LOGIN_AUTHENTICATION`: the signature file is ill-formed or the signature is invalid
  - `TEEC_ERROR_COMMUNICATION`:

    The Trusted Foundations is unreachable or has entered an unrecoverable “system panic” state.
  - `TEEC_ERROR_TARGET_DEAD`

    The service is in an unrecoverable error state (“panic”)
  - `TEEC_ERROR_BUSY`

    The service does not support multiple simultaneous sessions
  - `TEEC_ERROR_CANCEL`

    The operation was cancelled or timed out before it could reach the destination

- If the `returnOrigin` is equal to `TEEC_ORIGIN_TRUSTED_APP`, a return code defined by the protocol between the Client Application and the Secure Service.

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 with 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.

B3.3.5.7 TEEC_CloseSession

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

Description

This function closes a Session which has been opened with a Secure Service.

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

The Implementation does nothing if the `session` parameter is `NULL`.

The implementation of this function cannot fail: after this function returns the Client Application can always 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.

B3.3.5.8 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 Secure Service functions should be invoked. The supported command identifiers are defined by the Secure Service'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 Secure Service, 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 Input (or Inout) Memory References are synchronized from the Client Application to the Trusted Foundations.
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 Secure Service. During the execution of the Command, the Secure Service 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. When the command completes, the Implementation updates the `size` field of the Memory Reference structures flagged as output:
    - 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 Secure Service and synchronized by the Implementation.
    - 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 Secure Service. In these cases, this means that 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 this case, there is no guarantee that any data has been synchronized with the Client Application.

      This behavior is described in more detail in section B3.2.2.6, "*Variable Length Return Buffers*".

6. the Implementation updates 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 is one of the standard error codes and the error conditions are described below
- If the return origin is `TEEC_ORIGIN_TRUSTED_APP`, then the meaning of the return code is determined by the protocol exposed by the Secure Service. It is recommended that the Secure Service 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 Secure Service to invoke. The meaning of each Command Identifier must be defined in the protocol exposed by the Service.
- `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`, one of the following error codes:
  - `TEEC_ERROR_OUT_OF_MEMORY`:

    not enough resource is available to invoke the command
  - `TEEC_ERROR_COMMUNICATION`:

    The Trusted Foundations is unreachable or has entered an unrecoverable “system panic” state.
  - `TEEC_ERROR_TARGET_DEAD`

    The service is in an unrecoverable error state (“panic”)
  - `TEEC_ERROR_CANCEL`

    The operation was cancelled or timed out before it could reach the destination
- if the return origin is `TEEC_ORIGIN_TRUSTED_APP`, a return code defined by the Secure Service 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.

B3.3.5.9 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 Trusted Foundations 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 Secure Service 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 Secure Service. In these cases the return origin returned by `TEEC_OpenSession` or `TEEC_InvokeCommand` is either `TEEC_ORIGIN_API`, `TEEC_ORIGIN_COMMS`, or `TEEC_ORIGIN_TEE`, and the return code is `TEEC_ERROR_CANCEL`.

If the cancellation request is handled by the Secure Service itself then the return origin returned by `TEEC_OpenSession` or `TEEC_InvokeCommand` is `TEE_ORIGIN_TRUSTED_APP`, and the return code is defined by the Secure Service's protocol. If possible, Secure Services 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.

B3.3.5.10 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 B3-3: TF Client API Parameter Types.

The layout of the four parameter types within a 32-bit integer is the following:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU, set to 0																Type[3]				Type[2]				Type[1]				Type[0]			

Note that to maximize portability, this layout should be considered implementation-defined and the macro `TEEC_PARAM_TYPES` should always be used to build a value for the `paramTypes` field. However, the value 0 can always be used and is equivalent to all parameter types set to `TEEC_NONE`.

B3.3.5.11 TEEC_GETTIMELIMIT

```
void SMStubGetTimeLimit(
    TEEC_Context*     context,
    uint32_t          timeout,
    TEEC_TimeLimit*   timeLimit);
```

Description

Generates an absolute time limit from a relative timeout value.

The absolute time limit is equal to the current time plus the timeout specified in the parameter `timeout` expressed in milliseconds. Once a time limit has been retrieved, it can be used in the function `TEEC_OpenSessionEx` and `TEEC_InvokeCommandEx`.

A time limit is defined relative to the time origin associated with a particular TEE context. A pointer to a TEE contest must therefore be passed to the function.

This function is a non-standard extension to the Global Platform TEE Client API.

Parameters

- `context`: a pointer to an initialized TEE Context.
- `timeout`: the relative timeout in milliseconds.
- `timeLimit`: a pointer to a variable that receives the generated `TEEC_TimeLimit` structure.

B3.3.5.12 TEEC_OpenSessionEx

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

Description

This function is an extension to the GlobalPlatform TEE Client API. It works exactly as the function `TEEC_OpenSession` but takes an additional parameter `timeLimit`. This parameter must have been initialized with a call to `TEEC_GetTimeLimit`. After the specified time limit expires, the operation is automatically cancelled, as if `TEEC_RequestCancellation` was called.

Parameters

- `timeLimit`: The time limit after which the open operation is automatically cancelled if it has not already been completed. Setting this value to `NULL` indicates an infinite timeout. If `timeLimit` is not `NULL`, then `*timeLimit` must have been created by calling `TEEC_GetTimeLimit` in the TEE context `context`, otherwise the behavior is undefined.
- All other parameters: see the documentation of TEEC_OpenSession in section B3.3.5.6

B3.3.5.13 TEEC_INVOKECOMMANDEX

```
TEEC_Result TEEC_InvokeCommandEx(
    TEEC_Session*          session,
    const TEEC_TimeLimit* timeLimit,
    uint32_t               commandID,
    TEEC_Operation*        operation,
    uint32_t*              errorOrigin);
```

Description

This function is an extension to the GlobalPlatform TEE Client API. It works exactly as the function `TEEC_InvokeCommand` but takes an additional parameter `timeLimit`. This parameter must have been initialized with a call to `TEEC_GetTimeLimit`. After the specified time limit expires, the operation is automatically cancelled, as if `TEEC_RequestCancellation` was called.

Parameters

- `timeLimit`: The time limit after which the invoke operation is automatically cancelled if it has not already been completed. Setting this value to `NULL` indicates an infinite timeout. If `timeLimit` is not `NULL`, then `*timeLimit` must have been created by calling `TEEC_GetTimeLimit` in the same TEE context as the one used to open the session `session`, otherwise the behavior is undefined.
- All other parameters: see the documentation of TEEC_InvokeCommand in section B3.3.5.8

B3.3.5.14 TEEC_GetImplementationInfo

```
void TEEC_EXPORT TEEC_GetImplementationInfo(
   TEEC_Context*              context,
   TEEC_ImplementationInfo* description);
```

Description

This function can be used to retrieve information about the implementation into the structure `TEEC_ImplementationInfo` provided by the caller. The structure contains three fields which are filled with zero-terminated strings. The content of each string is implementation-defined but is intended to contain a version number, a build number, and a human-readable description of each of the following components:

- The field `apiDescription` contains a description of the user-space TF Client API implementation;
- The field `commsDescription` contains a description of the communication subsystem, which typically includes an OS-specific driver;
- The field `TEEDescription` contains a description of the Trusted Foundations Secure World.

This function may be called with `context` set to `NULL`, in which case it only sets the field `apiDescription`. The fields `commsDescription` and `TEEDescription` are set to an empty string in this case.

Parameters

- `context`: a pointer to a TEE context initialized with `TEEC_InitializeContext`. This parameter can be set to `NULL` to retrieve only the `apiDescription`.
- `description`: a pointer to a caller-allocated structure of type `TEEC_ImplementationInfo` filled by the implementation

B3.3.5.15 TEEC_GetImplementationLimits

```
void TEEC_EXPORT TEEC_GetImplementationLimits(
   TEEC_ImplementationLimits* limits
);
```

Description

This function can be used to retrieve some implementation limits related to shared memory. It fills the following fields of the structure `limits` of type TEEC_ImplementationLimits passed by the caller:

- `pageSize`: defines the size of a “page” in bytes. This is guaranteed to be a power of 2 (typically 1 byte, 4KB, 8KB, or 16KB).
- `tmprefMaxSize`: maximum size in bytes of a temporary memory reference. This must be understood for addresses rounded to whole page boundaries. To determine if a particular buffer can be passed as a temporary memory reference, use the following algorithm, where `buffer` is the buffer address and `size` the size in bytes of the buffer:

  ```
  // end address rounded up to the nearest page boundary
  (((uint32_t)buffer+size-1+pageSize) & (pageSize-1))
  -
  // start address rounded down to the nearest page boundary
  (((uint32_t)buffer) & (pageSize-1))
  <=
  nTmprefMaxSize
  ```

- `sharedMemMaxSize`: maximum size in bytes of a shared memory block that can be registered or allocated using the functions TEEC_RegisterSharedMemory or TEEC_AllocateSharedMemory. The same algorithm as for temporary memory references must be used to determine if a particular buffer can be registered.

Note that these are hard limits of the implementation. However, smaller buffers can these limits can fail to be processed due to run-time a low resource conditions.

Parameters

- `limits`: a pointer to a caller-allocated structure `TEEC_ImplementationLimits` filled by the implementation.

B3.3.5.16 TEEC_READSIGNATUREFILE

```
TEEC_Result TEEC_EXPORT TEEC_ReadSignatureFile(
   void**    ppSignatureFile,
   uint32_t* pnSignatureFileLength);
```

Description

Look for a signature file for the current application and read it entirely. The signature file is located in an implementation-defined way. For example, it can be a file in the same directory as the executable for the current application and has the additional extension '`.ssig`', but that may depend on the target Operating System.

This function is a helper and is not mandatory to use client authentication. Signature files can also be retrieved in other ways, for example by accessing other files, and passed in the parameter #3 of a `TEEC_OpenSession` operation.

On success, the implementation allocates a buffer of memory using `malloc` and fills it with the content of the signature file. A pointer to this buffer is returned to the caller in `*ppSignatureFile`. When no longer needed, it must be deallocated using the function `free`.

Parameters

- `ppSignatureFile`: Pointer to a variable of type `void*`. On success, this variable is filled with a pointer to the signature file allocated using the function `malloc`. On error, this variable is set to `NULL`.
- `pnSignatureFileLength`: Pointer to a variable of type `uint32_t`. On success, this variable is set with the number of bytes in the signature file. On error, it is set to 0.

Return Values

- `TEEC_SUCCESS` in case of success
- `TEEC_ERROR_OUT_OF_MEMORY`: not enough memory to allocate the copy of the signature file
- `TEEC_ERROR_ITEM_NOT_FOUND`: the signature file could not be found
- `TEEC_ERROR_OS`: the file was found but couldn't be read

B3.3.6 SIGNATURE FILE FORMAT SPECIFICATION

This section specifies the format of the signature file accepted by the TF Client API. It precisely defines what a well-formed signature file is. This specification is a series of "MUST" requirements. Whenever one of these requirements is violated, it means that the signature file is invalid.

A signature file is a manifest signed by a chain of certificates:

- The overall structure of the file is based on the PKCS#7 standard, **[PKCS#7]**. This structure is fully specified in section B3.3.6.1.
- The manifest is a textual description of the client properties, including the client identifier, which is a UUID, and client authentication data. The full specification of the manifest is provided in sectionB3.3.6.2.
- The chain of certificates embedded in the signature file is based on X509v3 certificates as defined in **[RFC3280]**. The present specification provides additional details to the RFC3280, and is fully specified in section B3.3.6.3.

B3.3.6.1 OVERALL FILE STRUCTURE

A signature file MUST be:

- A signed-data content as specified in the PKCS#7 standard (**[PKCS#7]**, Section 9).
- Encoded in DER. This means that all the data and certificates within the file MUST also be encoded in DER.

The ASN.1 syntax of a PKCS#7 SignedData object is recalled here for clarity:

```
SignedData ::= SEQUENCE {
  version           Version,
  digestAlgorithms DigestAlgorithmIdentifiers,
  contentInfo       ContentInfo,
  certificates [0] IMPLICIT   ExtendedCertificatesAndCertificates
OPTIONAL,
  crls          [1] IMPLICIT   CertificateRevocationLists OPTIONAL,
  signerInfos       SignerInfos
}

DigestAlgorithmIdentifiers ::=  SET OF DigestAlgorithmIdentifier

ContentInfo ::= SEQUENCE {
  contentType       ContentType,
  content       [0] EXPLICIT ANY DEFINED BY contentType OPTIONAL
}

ContentType ::= OBJECT IDENTIFIER

SignerInfos ::= SET OF SignerInfo

SignerInfo ::= SEQUENCE {
  version                    Version,
  issuerAndSerialNumber      IssuerAndSerialNumber,
  digestAlgorithm            DigestAlgorithmIdentifier,
  authenticatedAttributes    [0] IMPLICIT   Attributes OPTIONAL,
  digestEncryptionAlgorithm DigestEncryptionAlgorithmIdentifier,
  encryptedDigest            EncryptedDigest,
  unauthenticatedAttributes [1] IMPLICIT   Attributes OPTIONAL
}

EncryptedDigest ::= OCTET STRING
```

```
IssuerAndSerialNumber ::= SEQUENCE {
  issuer       Name,
  serialNumber CertificateSerialNumber
}
```

Compared to the PKCS#7 standard, the present specification puts additional constraints on the fields of the structure:

Table B3-5: TF Client API: Structure of a Client Signature File

Field		Value/Comment
`version`		This is the syntax version number and MUST have a value of “1”.

`digestAlgorithms`		This field MUST contain a single value which MUST be the object identifier for SHA1:
		`sha-1 OBJECT IDENTIFIER ::= {`
		`  iso(1)`
		`  identifiedorganization(3)`
		`  oiw(14)`
		`  secsig(3)`
		`  algorithm(2) 26`
		`}`

`contentInfo`		This is the signed content, and consists of a content type identifier and the content itself.

	`contentType`	The `contentType` MUST be data, i.e.:
		`data OBJECT IDENTIFIER ::= {`
		`  iso(1)`
		`  member-body(2)`
		`  us(840)`
		`  rsadsi(113549)`
		`  pkcs(1)`
		`  pkcs-7(7) 1`
		`}`

	`content`	The `content` field MUST be a non-NULL DER OCTET STRING containing the magic number `{0x53, 0x4D, 0x41, 0x4E}` on the four first bytes followed by the manifest formatted as specified in section B3.3.6.2
`certificates`		This field MAY contain a set of certificates that validates each certificate up to the signer’s certificate. The certificates MUST comply with the specification provided in section B3.3.6.3
`crls`		This field MUST NOT be present
`signerInfos`		There MUST be one and only one signer. The fields of this structure MUST be as follows:
	`version`	This is the syntax version number and MUST have a value of “1”.
	`issuerAndSerialNumber`	This field identifies the signer’s certificate by using issuer *dName* and certificate serial number. If the signature file contains a chain of certificates, this MUST match the issuer and serial number of the last certificate in the chain (signer’s certificate)

	`digestAlgorithm`	This MUST be the SHA1 object identifier:
		`sha-1 OBJECT IDENTIFIER ::= {`
		`  iso(1)`
		`  identifiedorganization(3)`
		`  oiw(14)`
		`  secsig(3)`
		`  algorithm(2) 26`
		`}`

	`authenticatedAttributes`	This field MUST NOT be present.

	`digestEncryptionAlgorithm`	This field MUST be the RSA object identifier:
		`rsaEncryption OBJECT IDENTIFIER ::= {`
		`  iso(1)`
		`  member-body(2)`
		`  us(840)`
		`  rsadsi(113549)`
		`  pkcs(1)`
		`  pkcs-1(1) 1`
		`}`

	`encryptedDigest`	This is the actual signature value produced using the signer's private key. Given the object identifiers required in `digestAlgorithm` and `digestEncryptionAlgorithm`, it MUST therefore have been generated: • by hashing the content using the SHA-1 algorithm. The whole DER OCTET STRING of content MUST be hashed. • by then signing the digest using the RSASSA-PKCS1-v1_5 scheme as defined in **[PKCS#1]** and the signer's private key.
	`unauthenticatedAttributes`	This field MUST not be present.

B3.3.6.2 Manifest File Format

This section specifies the format of the manifest file, i.e., the content field of the PKCS#7 signature file. A manifest file represents a set of properties. Each property has a name, which is a Unicode character string, and a value, which is also a Unicode character string. Some properties have predefined meanings, whereas the others can be used by the authority to describe some characteristics of the client, for example, some of its rights.

The manifest file MUST be a Unicode text file encoded in UTF-8, as specified in **[RFC3629]** In particular, the manifest file MUST NOT end in the middle of a UTF-8 sequence. The string of Unicode characters MUST comply with the following BNF syntax:

```
property-file:   +(property newline) ?property
property:        name COLON *space value *space
newline:         CR LF | LF | CR
space:           SPACE | TAB
name:            alphanum *headerchar
value:           *otherchar
alphanum:        {A-Z} | {a-z} | {0-9}
headerchar:      alphanum | MINUS | UNDERSCORE | DOT
otherchar:       any UTF-8 character except NUL, CR and LF
```

```
CR:                 <Unicode CR (U+000D)>
LF:                 <Unicode LF (U+000A)>
SPACE:              <Unicode SPACE (U+0020)>
TAB:                <Unicode TAB (U+0009)>
NUL:                <Unicode NUL (U+0000)>
MINUS:              <Unicode MINUS (U+002D)>
UNDERSCORE:         <Unicode UNDERSCORE (U+005F)>
DOT:                <Unicode DOT (U+002E)>
COLON:              <Unicode COLON (U+003A)>
```

Comments on BNF Syntax

The manifest file is a list of properties, where each property:

- Is a name-value pair, where the value is separated from the property by a colon character.
- Is separated from other properties by a single newline.
- Has a name that must start with a letter or a digit and can continue with numbers, digits, dashes '-', underscores '_' and dots '.'.
- Has a value that can contain any characters, except newlines and NUL. Note that the value is stripped of any leading or trailing spaces and tabs by the implementation when read.

There can optionally be a trailing newline at the end of the file.

If several properties have the same name, only the value of the first instance of that property, starting from the beginning of the manifest is used. For the Trusted Foundations, the other values are not used, and discarded.

The manifest file MUST contain a property named "`sm.client.id`". Its value MUST be the valid string representation of a Universally Unique Identifier as defined in **[RFC4122]**, which MUST NOT be interpreted in a way which is case sensitive. This is the identifier of the client application.

The manifest MUST contain a property named "`sm.client.exec.hash`" which is the client authentication data.The precise definition of the value of this property is implementation-defined and depend on the Operating System. It may a digest of the application executable or may rely on an existing application identifier provided by the OS. When the signature file is processed, the Trusted Foundations computes the client authentication data corresponding to the calling application and compares the result to the value of the `sm.client.exec.hash` property.

Note that the value of the `sm.client.exec.hash` property is not necessarily a hash.

The manifest MUST NOT contain any other properties whose name starts with the "`sm.client.`" prefix. Note that the manifest may contain application-dependent properties but these must not start with the "`sm.client`" prefix.

Here is an example of a correct manifest file containing two application-specified properties ("Name" and "Version"):

```
sm.client.id: 800730f9-cf5d-43d5-898f-6e5940bcaebb <CR> <LF>
Name: Media Player <CR> <LF>
Version: 1.0 <CR> <LF>
sm.client.exec.hash: vcHUT2iGKzzK8dHpOT1q1Q==
```

B3.3.6.3 Certificates Format and Constraints

A signature file format always contains a signature of the manifest file in the `encryptedDigest` field of the PKCS#7 structure.

This signature can be either directly checked by a root public key in the Trusted Foundations or make use of a certificate chain put in the field `certificates`.

If the `certificates` field is present, it contains a set of certificates. The order of the elements in this set is not relevant. Each certificate MUST comply with the X509v3 specification **[RFC3280]**. The relevant ASN1 syntax is recalled here for clarity:

```
Certificate ::= SEQUENCE {
  tbsCertificate       TBSCertificate,
  signatureAlgorithm   AlgorithmIdentifier,
  signatureValue       BIT STRING
}

TBSCertificate ::= SEQUENCE {
  version         [0]  EXPLICIT Version DEFAULT v1,
  serialNumber         CertificateSerialNumber,
  signature            AlgorithmIdentifier,
  issuer               Name,
  validity             Validity,
  subject              Name,
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                       -- If present, version MUST be v2 or v3
  subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                       -- If present, version MUST be v2 or v3
  extensions      [3]  EXPLICIT Extensions OPTIONAL
                       -- If present, version MUST be v3
}

Version ::= INTEGER { v1(0), v2(1), v3(2) }

CertificateSerialNumber ::= INTEGER

AlgorithmIdentifier ::= SEQUENCE {
  algorithm            OBJECT IDENTIFIER,
  parameters           ANY DEFINED BY algorithm OPTIONAL
}

Validity ::= SEQUENCE {
  notBefore            Time,
  notAfter             Time
}

Time ::= CHOICE {
  utcTime              UTCTime,
  generalTime          GeneralizedTime
}

UniqueIdentifier ::= BIT STRING

SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm            AlgorithmIdentifier,
  subjectPublicKey     BIT STRING
}

Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension

Extension ::= SEQUENCE {
  extnID               OBJECT IDENTIFIER,
  critical             BOOLEAN DEFAULT FALSE,
  extnValue            OCTET STRING
}
```

Each certificate in the set must comply with additional constraints introduced by the present specification:

Table B3-6: TF Client API: Client Certificate Format

Field	Value/Comment
`tbsCertificate`	
`version`	Value MUST be 2 (X.509 v3 certificate).
`serialNumber`	A Unique Integer (not interpreted by the Trusted Foundations).
`signature`	MUST be RSA-SHA1, i.e.:
`algorithm`	OID of the signature algorithm. This field MUST be iso.member-body.US.rsadsi.pkcs.pkcs-1.sha1WithRSAEncryption (OID 1:2:840:113549:1:1:5)
`parameters`	MUST be present and MUST be NULL, as specified in **[PKCS#1]**.
`issuer`	The certificate issuer Distinguished Name.
`validity`	The certificate validity period.
`subject`	The certificate Distinguished Name.
`subjectPublicKeyInfo`	MUST contain a RSA public key:
`algorithm`	MUST specify RSA, i.e.:
`algorithm`	MUST be iso.member-body.US.rsadsi.pkcs.pkcs-1.rsaEncryption (OID 1:2:840:113549:1:1:1).
`parameters`	MUST be NULL.
`subjectPublicKey`	The actual `subjectPublicKey` field MUST contain the certificate public key as a BIT STRING. This MUST be a `RSAPublicKey` object encoded in DER.
`issuerUniqueID`	MUST not be present.
`subjectUniqueID`	MUST not be present.
`extensions`	See below.
`signatureAlgorithm`	MUST be iso.member-body.US.rsadsi.pkcs.pkcs-1.sha1WithRSAEncryption (OID 1:2:840:113549:1:1:5). In other words, certificates not using the RSASSA-PKCS1-v1_5 signature scheme are not supported.
`signatureValue`	MUST be the signature by the issuer's private key of the whole tbsCertificate field encoded in DER. See below for an explanation of which key must be used.

The following constraints must be satisfied for a signature file to be considered as valid:

The PKCS#7 `content` field (the manifest) MUST be signed by either the root public key installed in the Trusted Foundations or there must exist a chain of certificates in the `certificates` set with the following properties:

1. The `content` field must be signed by the public key of the first certificate in the chain. This certificate is called the "signer's certificate". All other certificates in the chain are called "CA's certificates";
2. The `issuerAndSerialNumber` field in the PKCS#7 structure MUST match the distinguished name (`subject` field) and serial number of the signer's certificate;
3. For each certificate in the chain but the last, the distinguished name of the certificate issuer (`issuer` field) MUST be equal to the distinguished name (`subject` field) of the next certificate in the chain;
4. For each certificate in the chain but the last, the signature of the certificate MUST be verifiable by the public key of the next certificate in the chain;

5. The signature of the last certificate in the chain MUST be verifiable by the root public key installed in the Trusted Foundations;
6. If the last certificate in the chain is self-issued, i.e., if `subject` is equal to `issuer`, then it MUST be self-signed, i.e. its signature must be verifiable by its own public key;
7. Each certificate in the list must contain some extensions and constraints as specified in **[RFC3280]**:
    - the system processing the signature file (and so, the certificates) will reject a certificate if it encounters a critical extension it does not recognize. In the present specification, only two extensions are supported: `BasicContraints` and `KeyUsage`. Therefore, a certificate MUST NOT contain an extension marked as critical if it is not described in the present specification;

For the sake of clarity, the definition of the two supported constraints is recalled here:

```
id-ce   OBJECT IDENTIFIER ::=  { joint-iso-ccitt(2) ds(5) 29 }

id-ce-basicConstraints OBJECT IDENTIFIER ::= { id-ce 19 }

BasicConstraints ::= SEQUENCE {
  cA                        BOOLEAN DEFAULT FALSE,
  pathLenConstraint         INTEGER (0..MAX) OPTIONAL
}

id-ce-keyUsage OBJECT IDENTIFIER ::=  { id-ce 15 }

KeyUsage ::= BIT STRING {
  digitalSignature          (0),
  nonRepudiation            (1),
  keyEncipherment           (2),
  dataEncipherment          (3),
  keyAgreement              (4),
  keyCertSign               (5),
  cRLSign                   (6),
  encipherOnly              (7),
  decipherOnly              (8)
}
```

- The following constraint applies to the signer's certificate:

Signer's certificate Extensions	
`BasicConstraints`	This extension is optional.
`KeyUsage`	This extension MUST be present and the key usage must be contain the `digitalSignature` (0) bit

- The following constraint applies to any of the CA's certificate:

CA's certificate Extensions	
`BasicConstraints`	This extension MUST be present and the `cA` field MUST be set to TRUE
`KeyUsage`	This extension MUST be present and the key usage must be contain the `keyCertSign` (5) bit

Note that a certificate can possibly meet the constraints to be both a signer's certificate and CA certificate, i.e., have the `cA` field asserted and, at the same time, have both the `digitalSignature` bit and *keyCertSign* bit set. Such a certificate can play both roles in different signature files.

All products support RSA keys up to 2048 bits in length for all the keys in the chain. Some implementations may support longer keys; consult the *Product Reference Manual* shipped with your product for details.

Chapter B4 TRUSTED FOUNDATIONS CRYPTOGRAPHIC API

This chapter is the specification of the Cryptographic API, used both as an External API for the Normal World and as an Internal API as part of SSDI for the Secure World.

B4.1 About the Trusted Foundations Cryptographic API

This chapter is the specification of the Trusted Foundations' Cryptographic API.

The Cryptographic API is based on a subset of RSA Security's public PKCS#11 API specification **[PKCS#11]**, also known as "Cryptoki"[1], which provides a portable and standard programming interface to underlying cryptographic providers, such as software crypto libraries, smart cards, hardware cryptographic accelerators, or the Trusted Foundations.

The scope of the Cryptographic API is only to provide access to cryptographic functionalities such as hashing, symmetric and asymmetric key algorithms, key handling and data storage. It is up to applications, services, or libraries built on top of the Cryptographic API to provide higher level functionalities such as parsing of ASN.1, managing of X.509, SSL protocol stack, or DRM.

This API is available both to Applications executing in the Normal World and to Secure Services executing in the Secure World.

Trusted Foundations Cryptographic API includes only a subset of the general Cryptoki standard and adds some proprietary features. Sometimes the behavior of the API also deviates from the standard. The differences between the standard Cryptoki and the Cryptographic API are highlighted in section B4.2, "*Differences with Cryptoki*".

The other sections are the complete reference for the Cryptographic API:

- Section B4.3, "*Data types and notations*";
- Section B4.4, "*Objects and Attributes*";
- Section B4.5, "*Functions*";
- Section B4.6, "*Cryptoki Mechanisms*".

[1] This chapter contains material extracted from the "RSA Security Inc. Public-Key Cryptography Standards (PKCS) #11" **[PKCS#11]**.

B4.2 DIFFERENCES WITH CRYPTOKI

The **[PKCS#11]** specification contains many options; typically implementations of the standard do not support all features and algorithms that the standard defines. This section documents the choice of functionalities offered by this Cryptographic API compared to the full reference Cryptoki.

Note that a specific Trusted Foundations product that implements this Cryptographic API may provide only a subset of the functionalities of the API. In particular, an implementation may not provide all the algorithms, mechanisms, or key sizes described in this specification. A particular integration of a given product may also further restrict the supported functionalities. Refer to the *Product Reference Manual* shipped with your product for details of the cryptographic support provided.

This API is available both for Applications executing in the Normal World and for Secure Services executing in the Secure World. The two versions of the API occasionally behave slightly differently. When this is the case, the differences are clearly highlighted in this chapter.

The Cryptographic API differs from the standard Cryptoki on the following points:

- The API provides access to “virtual tokens”, not physical token. A virtual token is a storage space created on demand and associated to a certain caller identity. For example, there is a different virtual token created for each application and service accessing the API and these virtual tokens are separated. See section B4.2.1, “*Slot Identifiers*, Virtual Tokens, ”;
- Sessions connect a caller to a virtual token and the API introduces some restrictions on how sessions can be used and how they interact with each other. This is described in section B4.2.2, “*Sessions*”
- Only a subset of all the Cryptoki object types is supported. The attributes that an object can have are also restricted. See section B4.2.3, “*Supported Objects and Attributes*”;
- Only a subset of all the Cryptoki functions and mechanism is supported. See section B4.2.4, “*Supported Cryptoki Functions*”;
- Finally, the API introduces some non-standard functions, attributes, and mechanisms described in section B4.2.6, “*Non-Standard Extensions*”

B4.2.1 SLOT IDENTIFIERS, VIRTUAL TOKENS, AND TOKEN STORAGES

The Cryptographic API provides access to a number of slots that each contain a virtual non-removable token. When an application or service calls the function **C_OpenSession**, the system determines a virtual token. This token depends on the calling world, the identity of the caller, and the slot identifier.

Virtual Tokens for the Normal World

For applications calling from the Normal World, the virtual tokens accessible are the following:

- Tokens that are shared among all the Normal World applications;
- Tokens that are private to the calling application, different applications having access to different virtual tokens, even if they use the same slot identifier;
- Tokens that are shared among all applications that belong to a given “OS group”. This allows different applications to share objects.

The way applications are differentiated and the notion of OS group depends on the TF Client API implementation and therefore ultimately on the specific OS integration of your product. See the documentation of the function C_OpenSession in section B4.5.4.1 for more details.

Virtual Tokens for the Secure World

For services calling the API from the Secure World, the only virtual tokens accessible are tokens that are private to the calling service. Different services have access to different private tokens.

Note that virtual tokens for normal world applications are always separated from any virtual tokens for secure world services.

System vs. User Token Storages

For each kind of tokens listed above, there are several variants stored in different physical storage spaces. This version of the API supports two distinct token storages:

- The system storage
- The user storage

When an application or service opens a session on a virtual token, the slot identifier is used both to identify which kind of tokens is requested (shared, private, group), but which token storage to use.

Different token storages may use different physical storage spaces (partitions) and may have different life-cycles, but this is *implementation-dependent*.

Note that tokens of the same kinds but stored in different storages are always logically separated even if your product does not store them in physically separated areas.

B4.2.2 SESSIONS

The Cryptographic API introduces the following restrictions concerning the usage of Cryptoki sessions, operations, and objects.

B4.2.2.1 SESSION ISOLATION AND SECONDARY SESSIONS

Contrary to standard Cryptoki, the sessions opened by a caller of the API (also called "primary sessions") are completely independent. Object handles created or opened in a first session cannot be used in a second session. Furthermore, session objects, i.e., objects with the attribute `CKA_TOKEN` not set, or set to `CK_FALSE` are strictly attached to the session in which they are created. They cannot be used within another session and are automatically destroyed when the session is closed.

However, this Cryptographic API introduces an extension to the standard Cryptoki, called "secondary sessions". A secondary session is opened within an already opened primary session. Operations within secondary sessions can use object handles opened in their primary session. This allows sharing object handles among multiple parallel operations performed in different secondary sessions.

Secondary sessions can be used only for cryptographic or find operations, not for functions that can create or delete objects.

A primary session can be either Read-Only or Read-Write. Read-Only sessions allow the manipulation of session objects, but not of token objects. Read-Write sessions allow the creation and deletion of token objects. You can open multiple Read-Write or Read-Only sessions on a given token, even from different calling processes. The creation or deletion of a token object is immediately visible on all the sessions currently opened on the token.

B4.2.2.2 SO AND PIN NOT SUPPORTED

The Cryptographic API does not support the SO (Security Officer) type of user and performs an implicit login when a virtual token is opened.

When a session is opened, it is implicitly in the state "R/O User functions" or "R/W User functions" and has access to all objects of its virtual token. All objects are implicitly "private". The function **C_Login** and **C_Logout** are provided for compatibility reasons, but do nothing.

B4.2.2.3 LIMITED PARALLEL OPERATIONS

Many of the Cryptoki operations require multiple calls to the API and maintain an internal operation state. This kind of "stateful" operations are those initiated by one of the following functions:

- `C_FindObjectsInit`
- `C_EncryptInit`
- `C_DecryptInit`
- `C_DigestInit`
- `C_SignInit`
- `C_VerifyInit`

The stateful cryptographic operations can take their input data either in a single part or in multiple parts. In both cases, the operation always starts with a call to `C_<Op>Init` where the operation mechanism, keys, and parameters are specified. The caller can then either call `C_<Op>`, which terminates the operation with a single-part data, or call `C_<Op>Update` one or multiple times to provide the multi-part input data and finish the operation with `C_<Op>Final`. For a "find" operation, there is no "single-part" function.

A stateful operation is said to be "active" until the corresponding "final" or "single-part" function is called.

In the TF Cryptographic API, at a given time, there can be at most one active stateful operation in a given session, either primary or secondary. However, multiple stateful operations can be active at a given time if they belong to different sessions. In particular, parallel stateful operations can be initiated in several secondary sessions opened within the same primary session, which allows using the same key handles in these secondary sessions.

Note also that it is not possible to mix stateful operations of different kinds; for example, to start an encryption operation at the same time as a signature operation in the same session: you must use secondary sessions for this.

When a stateful operation is currently active within a session, the application or service can still use any other functions that are not stateful, like `C_CreateObject`, `C_DeriveKey`, `C_GenerateRandom`, etc.

Finally, note that the following functions are not supported:

- Dual-function cryptographic functions (`C_DigestEncryptUpdate`, `C_DecryptDigestUpdate`, etc.)
- Operation state saving and restoring functions (`C_GetOperationState`, `C_SetOperationState`)

B4.2.3 Supported Objects and Attributes

The Cryptographic API supports only data and key objects. Table B4-1 lists the types of objects supported by the Cryptographic API.

Table B4-1: Supported Cryptoki Objects

Object Types	Support
`CKO_DATA`	Supported
`CKO_SECRET_KEY`	Supported with the following key types: • AES (`CKK_AES`) • DES (`CKK_DES`) • Double-Length DES (`CKK_DES3`) • Triple-Length DES (`CKK_DES2`) • RC4 (`CKK_RC4`) • Generic secrets (`CKK_GENERIC_SECRET`): used for HMAC and vendor-defined mechanisms
`CKO_PUBLIC_KEY`	Supported with the following key types : • RSA (`CKK_RSA`) • DSA (`CKK_DSA`) • Diffie-Hellman (`CKK_DH`)
`CKO_PRIVATE_KEY`	

B4.2.3.1 Objects are Not Modifiable

Objects are not modifiable: once they are created, their attributes cannot be changed. The function `C_SetAttributeValue` is therefore not included in the Cryptographic API. As a consequence, from its creation, an object is either a token object or a session object and cannot change its status.

The API supports a limited form of object copying though. For example, you can duplicate a session object into a token object to make it persistent. See the function `C_CopyObject` (section B4.5.5.2) for more details.

B4.2.3.2 Limited Find Capabilities and Specific Identification Attributes

Contrary to standard Cryptoki where a find operation can be used on any of the object attributes and can match both token and session objects, the TF Cryptographic API introduces the following limitations:

- A find operation can only be used to search for token objects. Session objects are never returned by the `C_FindObjects` function;
- A find operation can only match the value of the attributes `CKA_ID` and `CKA_CLASS`;
- The attribute `CKA_ID` is mandatory for all token objects, including objects of the class `CKO_DATA`
- The combination of the attributes `CKA_ID` and `CKA_CLASS` must be unique among all token objects within the token: this combination is used as a primary index within the token
- The value of the `CKA_ID` attribute is limited to a maximum of 64 bytes

Note that, as a consequence, there are only four kinds of find operations:

- Lookup of a specific token object by specifying both the `CKA_CLASS` and `CKA_ID` attributes in the template passed to `C_FindObjectsInit`. This kind of find operation is guaranteed to return at most one object handle;
- Enumeration of all token objects having a given `CKA_ID`. This is guaranteed to return at most four object handles, because there are only four object classes supported;
- Enumeraton of all token objects in a certain class
- Enumeration of all the token objects in the token

B4.2.3.3 Supported Attributes

See section B4.4, “*Objects and Attributes*” for a complete reference on all supported attributes.

B4.2.4 Supported Cryptoki Functions

The Table B4-2 lists the Cryptoki functions included in this specification.

Table B4-2: Supported Cryptoki Functions

Category	Function	Description
General purpose functions	`C_Initialize`	Initializes the Cryptoki library for use by an application
	`C_Finalize`	Closes the library, and cleans up miscellaneous Cryptoki-associated resources
	`C_GetInfo`	Obtains general information about the Cryptoki implementation
Session management functions	`C_OpenSession`	Opens a connection between an application and a particular virtual token
	`C_CloseSession`	Closes a session
	`C_Login`	Does nothing
	`C_Logout`	Does nothing
Object management functions	`C_CreateObject`	Creates an object
	`C_DestroyObject`	Destroys an object
	`C_CopyObject`	Creates a copy of an existing object
	`C_GetAttributeValue`	Obtains the value of some attributes of an object
	`C_FindObjectsInit`	Initializes an object search operation

Category	Function	Description
	`C_FindObjects`	Continues an object search operation
	`C_FindObjectsFinal`	Finishes an object search operation
	`C_CloseObjectHandle`	Closes an object handle (this is a non-standard extension to Cryptoki)
Encryption functions	`C_EncryptInit`	Initializes an encryption operation
	`C_Encrypt`	Encrypts single-part data
	`C_EncryptUpdate`	Continues a multiple-part encryption operation
	`C_EncryptFinal`	Finishes a multiple-part encryption operation
Decryption functions	`C_DecryptInit`	Initializes a decryption operation
	`C_Decrypt`	Decrypts single-part encrypted data
	`C_DecryptUpdate`	Continues a multiple-part decryption operation
	`C_DecryptFinal`	Finishes a multiple-part decryption operation
Message digesting functions	`C_DigestInit`	Initializes a message-digesting operation
	`C_Digest`	Digests single-part data
	`C_DigestUpdate`	Continues a multiple-part digesting operation
	`C_DigestFinal`	Finishes a multiple-part digesting operation
Signing and MACing functions	`C_SignInit`	Initializes a signature operation
	`C_Sign`	Signs single-part data
	`C_SignUpdate`	Continues a multiple-part signature operation
	`C_SignFinal`	Finishes a multiple-part signature operation
Functions for verifying signatures and MACs	`C_VerifyInit`	Initializes a verification operation
	`C_Verify`	Verifies a signature on single-part data
	`C_VerifyUpdate`	Continues a multiple-part verification operation
	`C_VerifyFinal`	Finishes a multiple-part verification operation
Key management functions	`C_GenerateKey`	Generates a secret key
	`C_GenerateKeyPair`	Generates a public-key/private-key pair
	`C_DeriveKey`	Derives a key from a base key
Random number generation functions	`C_SeedRandom`	Mixes in additional seed material to the random number generator
	`C_GenerateRandom`	Generates random data

B4.2.5 SUPPORTED MECHANISMS

The mechanisms supported by this specification are listed in Table B4-3 and are fully specified in section B4.6, "*Cryptoki Mechanisms*".

Note, however, that a specific Trusted Foundations product may not provide all the algorithms, mechanisms, or key sizes described in this specification. An integration of the product may also further restrict the supported functionalities.

Refer to the *Product Reference Manual* shipped with your product for details of the cryptographic support provided with your product.

Table B4-3: Supported Cryptoki Mechanisms

Mechanism	Encrypt& Decrypt	Sign & Verify	Digest	GenerateKey / KeyPair	Derive
CKM_RSA_PKCS_KEY_PAIR_GEN				✓	
CKM_RSA_PKCS	✓ (single part only)				
CKM_RSA_PKCS_OAEP	✓ (single part only)				
CKM_RSA_PKCS_PSS		✓ (single part only)			
CKM_RSA_X_509	✓ (single part only)	✓ (single part only)			
CKM_MD5_RSA_PKCS		✓			
CKM_SHA1_RSA_PKCS		✓			
CKM_SHA1_RSA_PKCS_PSS		✓			
CKM_SHA224_RSA_PKCS		✓			
CKM_SHA224_RSA_PKCS_PSS		✓			
CKM_SHA256_RSA_PKCS		✓			
CKM_SHA256_RSA_PKCS_PSS		✓			
CKM_SHA384_RSA_PKCS		✓			
CKM_SHA384_RSA_PKCS_PSS		✓			
CKM_SHA512_RSA_PKCS		✓			
CKM_SHA512_RSA_PKCS_PSS		✓			
CKM_AES_KEY_GEN				✓	
CKM_AES_ECB	✓				
CKM_AES_CBC	✓				
CKM_AES_CTR	✓				
CKM_DES_KEY_GEN				✓	
CKM_DES_ECB	✓				
CKM_DES_CBC	✓				
CKM_DES2_KEY_GEN				✓	
CKM_DES3_KEY_GEN				✓	
CKM_DES3_ECB	✓				
CKM_DES3_CBC	✓				
CKM_RC4	✓				
CKM_RC4_KEY_GEN				✓	
CKM_MD5			✓		
CKM_SHA_1			✓		
CKM_SHA224			✓		
CKM_SHA256			✓		
CKM_SHA384			✓		
CKM_SHA512			✓		
CKM_DSA_SHA1		✓			
CKM_DSA		✓ (single-part only)			
CKM_DSA_KEY_PAIR_GEN				✓	
CKM_DH_PKCS_KEY_PAIR_GEN				✓	
CKM_DH_PKCS_DERIVE					✓ (with limitations)
CKM_MD5_HMAC		✓			
CKM_SHA_1_HMAC		✓			
CKM_SHA224_HMAC		✓			
CKM_SHA256_HMAC		✓			
CKM_SHA348_HMAC		✓			
CKM_SHA512_HMAC		✓			
CKM_GENERIC_SECRET_KEY_GEN				✓	
CKMV_IMPLEMENTATION_DEFINED_*X*(X=0 to 15)	✓				
CKMV_AES_CTR ***(deprecated)***	✓				

The mechanisms `CKMV_IMPLEMENTATION_DEFINED_`*`X`* and `CKMV_AES_CTR` are non-standard extensions.

B4.2.6 Non-Standard Extensions

The Cryptographic API introduces the following non-standard extensions to Cryptoki:

- All token objects are uniquely identified by a primary index formed by the combination of the attribute `CKA_ID` and `CKA_CLASS`. The attribute `CKA_ID` is mandatory on all token objects, not only key objects. Its value is limited to a maximum of 64 bytes;
- There are 16 mechanisms, named `CKMV_IMPLEMENTATION_DEFINED_0` to `CKMV_IMPLEMENTATION_DEFINED_15`, that provide access to product-specific or integration-specific cryptographic block ciphers. Consult the *Product Reference Manual* of your product to know how to use these these mechanisms. See section B4.6.15, "*Implementation-Defined Block Ciphers*" for more details;
- Generic secret keys (`CKK_GENERIC_SECRET`) can be used for signature and verification operations (HMAC) and for encryption and decryption operations with the implementation-defined block ciphers;
- The attribute `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY` is a non-standard boolean attribute. When it is set to true on a base key, the keys derived from the base key can be marked non-sensitive, i.e., to have their attribute `CKA_SENSITIVE` set to `CK_FALSE`. This allows Diffie-Hellman shared secret keys to be accessed in clear-text if necessary;

  Note that, by default, all secret and private keys are sensitive and cannot be read in clear-text. However, the Diffie-Hellman generated shared secret must typically be processed using a variety of derivation mechanisms that are not included in the Cryptographic API. So, there is a need to extract the shared secret to perform the derivation in the caller and there is also a need to control this access. See section B4.6.2.4, "*PKCS#3 Diffie-Hellman Key Derivation*" for more details.

  Note also that if your product implements the Developer APIs V3.1 or later, then RSA private keys can optionally be marked as non-sensitive too. See section B4.6.1.2, "*RSA private key objects*" for more details;
- In the **Internal Cryptographic API**, a service can establish a direct shortcut between commands received from its client and cryptographic update functions. This is described in section B2.3.14.1, "*Cryptoki Update Shortcut*";
- Secondary sessions can be opened to allow object handles to be part of multiple parallel stateful operations;
- An Object handle can be explicitely closed when it is no longer used and before the session terminates by using the functions `C_CloseObjectHandle` or `SHandleClose`. This allows a tighter control on the resource utilization;
- *The mechanism `CKMV_AES_CTR` is a vendor-defined mechanism to perform AES with Counter mode. It was introduced before the standard mechanism `CKM_AES_CTR` was defined in PKCS#11 v2.20 Amendment 3. `CKMV_AES_CTR` is now* ***deprecated*** *and you should use the standard mechanism `CKM_AES_CTR` instead. See section B4.6.4.6, "AES-Counter Mode (Non-Standard Mechanism -- Deprecated)" for more details.*

B4.3 DATA TYPES AND NOTATIONS

This section gives details of the Cryptoki types for representing data, as well as the prefix notation used when naming attributes and variables.

The details of the more complex data types used for holding parameters for the various mechanisms, and the pointers to those parameters, are not described here; these types are described with the information on the mechanisms themselves.

B4.3.1 PREFIX NOTATION

Table B4-4 gives the prefix notation is used in this standard. Upper case prefixes are used to describe types and constants, except in the case `C_` which is used to prefix function names. Lower case prefixes are used to indicate the type of data a variable represents.

Table B4-4: Cryptoki Prefixes

Prefix	Description
`C_`	Function
`CK_`	Data type or general constant
`CKA_`	Attribute
`CKF_`	Bit flag
`CKG_`	Mask generation function
`CKK_`	Key type
`CKM_`	Mechanism type
`CKO_`	Object class
`CKR_`	Return value
`CKU_`	User type
`CKZ_`	Salt/Encoding parameter source
`h`	a handle
`ul`	a `CK_ULONG`
`p`	a pointer
`pb`	a pointer to a `CK_BYTE`
`ph`	a pointer to a handle
`pul`	a pointer to a `CK_ULONG`

B4.3.2 BASIC TYPES

Cryptoki defines the following data types:

```
/* an unsigned 8-bit value */
typedef uint8_t CK_BYTE;

/* an unsigned 8-bit character */
typedef CK_BYTE CK_CHAR;

/* an 8-bit UTF-8 character */
typedef CK_BYTE CK_UTF8CHAR;
```

```
/* a BYTE-sized Boolean flag */
typedef bool CK_BBOOL;

/* an unsigned value, 32 bits long */
typedef uint32_t CK_ULONG;

/* a signed value, the same size as a CK_ULONG */
typedef int32_t CK_LONG;

/* 32 bits long; each bit is a Boolean flag */
typedef CK_ULONG CK_FLAGS;
```

In this API, the `CK_BBOOL` data type is an alias of `bool`. For convenience, the following preprocessor definitions are also provided:

```
#define CK_FALSE false
#define CK_TRUE  true
```

B4.3.3 OTHER CRYPTOKI TYPES

B4.3.3.1 VERSION AND INFO TYPES

B4.3.3.1.1 CK_VERSION

CK_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or an SSL implementation, or the hardware or firmware version of a slot or token. It is defined as follows:

```
typedef struct CK_VERSION {
  CK_BYTE major;
  CK_BYTE minor;
} CK_VERSION;
```

The fields of the structure have the following meanings:

- `major`: major version number (the integer portion of the version)
- `minor`: minor version number (the hundredths portion of the version)

Example: For version 1.0, *major* = 1 and *minor* = 0. For version 2.10, *major* = 2 and *minor* = 10.

B4.3.3.1.2 CK_INFO

CK_INFO provides general information about Cryptoki. It is defined as follows:

```
typedef struct CK_INFO {
  CK_VERSION cryptokiVersion;
  CK_UTF8CHAR manufacturerID[32];
  CK_FLAGS flags;
  CK_UTF8CHAR libraryDescription[32];
  CK_VERSION libraryVersion;
} CK_INFO;
```

The fields of the structure have the following meanings:

- `cryptokiVersion`: Cryptoki interface version number, for compatibility with future revisions of this interface.
- `manufacturerID`: ID of the Cryptoki library manufacturer. Must be padded with the blank character (‘ ’). Should *not* be null-terminated.
- `flags`: Bit flags reserved for future versions. Must be zero for this version.
- `libraryDescription`: Character-string description of the library. Must be padded with the blank character (‘ ’). Should *not* be null-terminated.
- `libraryVersion`: Cryptoki library version number.

For libraries written to this document, the value of `cryptokiVersion` should match the version of this document; the value of `libraryVersion` is the version number of the library software itself.

See section B4.5.3.3, “C_GetInfo” for more information.

B4.3.3.2 Slot and token types

Cryptoki represents slot with the following type:

B4.3.3.2.1 CK_SLOT_ID

`CK_SLOT_ID` is a Cryptoki-assigned value that identifies a slot. It is defined as follows:

```
typedef CK_ULONG CK_SLOT_ID;
```

See the documentation of the function C_OpenSession in section B4.5.4.1 for a description of the slot identifiers supported by the Cryptographic API.

B4.3.3.3 Session types

Cryptoki represents session information with the following type:

B4.3.3.3.1 CK_SESSION_HANDLE

`CK_SESSION_HANDLE` is a Cryptoki-assigned value that identifies a session. It is defined as follows:

```
typedef CK_ULONG CK_SESSION_HANDLE;
```

Valid session handles in Cryptoki always have nonzero values. For developers’ convenience, Cryptoki defines the following symbolic value:

```
#define CK_INVALID_HANDLE 0
```

B4.3.3.3.2 CK_USER_TYPE

`CK_USER_TYPE` holds the types of Cryptoki user. It is defined as follows:

```
typedef CK_ULONG CK_USER_TYPE;
```

In the Cryptographic API, the only type of user available is the following:

```
#define CKU_USER 1
```

B4.3.3.3.3 CK_SESSION_INFO

The following table defines the *flags* field used by `C_OpenSession`:

Table B4-5: Cryptoki Session Information Flags

Bit Flag	**Mask**	**Meaning**
`CKF_RW_SESSION`	`0x00000002`	True if the session is read/write; false if the session is read-only
`CKF_SERIAL_SESSION`	`0x00000004`	This flag is provided for backward compatibility, and should always be set to true
`CKVF_OPEN_SUB_SESSION`	`0x00000008`	This flag means that the function `C_OpenSession` is used to open a new secondary session within an existing primary session. See section B4.5.4.1 for more details

B4.3.3.4 Object types

Cryptoki represents object information with the following types:

B4.3.3.4.1 CK_OBJECT_HANDLE

`CK_OBJECT_HANDLE` is a token-specific identifier for an object. It is defined as follows:

```
typedef CK_ULONG CK_OBJECT_HANDLE;
```

When an object is created or found on a token, Cryptoki assigns it an object handle for that session to use to access it. A particular object on a token does not necessarily have a handle which is fixed for the lifetime of the object; however, if a particular session can use a particular handle to access a particular object, then that session will continue to be able to use that handle to access that object as long as the session continues to exist, the object continues to exist, and the object continues to be accessible to the session.

Valid object handles in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
#define CK_INVALID_HANDLE 0
```

An object handle is automatically closed when the session in which it has been created is closed.

An object handle can also be closed using the non-standard function `C_CloseObjectHandle`. In the Internal API, the function `SHandleClose` can also be used to close an object handle before the session is closed. Closing the object handle allows to release the associated resources sooner than at the end of the session.

B4.3.3.4.2 CK_OBJECT_CLASS

`CK_OBJECT_CLASS` is a value that identifies the classes (or types) of objects that Cryptoki recognizes. It is defined as follows:

```
typedef CK_ULONG CK_OBJECT_CLASS;
```

Object classes are defined with the objects that use them. The type is specified on an object through the CKA_CLASS attribute of the object.

B4.3.3.4.3 CK_KEY_TYPE

`CK_KEY_TYPE` is a value that identifies a key type. It is defined as follows:

```
typedef CK_ULONG CK_KEY_TYPE;
```

Key types are defined with the objects and mechanisms that use them. The key type is specified on an object through the `CKA_KEY_TYPE` attribute of the object.

B4.3.3.4.4 CK_ATTRIBUTE_TYPE

`CK_ATTRIBUTE_TYPE` is a value that identifies an attribute type. It is defined as follows:

```
typedef CK_ULONG CK_ATTRIBUTE_TYPE;
```

Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an object as a list of type, length value items. These are often specified as an attribute template.

B4.3.3.4.5 CK_ATTRIBUTE

`CK_ATTRIBUTE` is a structure that includes the type, value, and length of an attribute. It is defined as follows:

```
typedef struct CK_ATTRIBUTE {
  CK_ATTRIBUTE_TYPE type;
  void* pValue;
  CK_ULONG ulValueLen;
} CK_ATTRIBUTE;
```

The fields of the structure have the following meanings:

- `type`: the attribute type
- `pValue`: pointer to the value of the attribute
- `ulValueLen`: length in bytes of the value

If an attribute has no value, then `ulValueLen` = 0, and the value of `pValue` is irrelevant. An array of `CK_ATTRIBUTE`s is called a "template" and is used for creating, manipulating and searching for objects. The order of the attributes in a template *never* matters, even if the template contains vendor-specific attributes. Note that `pValue` is a "void" pointer, facilitating the passing of arbitrary values. The application must ensure that the pointer can be safely cast to the expected type.

B4.3.3.5 DATA TYPES FOR MECHANISMS

Cryptoki supports the following types for describing mechanisms and parameters to them:

B4.3.3.5.1 CK_MECHANISM_TYPE

`CK_MECHANISM_TYPE` is a value that identifies a mechanism type. It is defined as follows:

```
typedef CK_ULONG CK_MECHANISM_TYPE;
```

Mechanism types are defined with the objects and mechanism descriptions that use them.

See section B4.2.5, "*Supported Mechanisms*" for a definition of all the types of mechanisms supported by the Cryptographic API.

B4.3.3.5.2 CK_MECHANISM

`CK_MECHANISM` is a structure that specifies a particular mechanism and any parameters it requires. It is defined as follows:

```
typedef struct CK_MECHANISM {
  CK_MECHANISM_TYPE mechanism;
  void* pParameter;
  CK_ULONG ulParameterLen;
} CK_MECHANISM;
```

The fields of the structure have the following meanings:

- `mechanism`: the type of mechanism
- `pParameter`: pointer to the parameter if required by the mechanism
- `ulParameterLen`: length in bytes of the parameter

Note that `pParameter` is a "void" pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

B4.3.3.6 FUNCTION TYPES

Cryptoki represents information about functions with the following data types:

B4.3.3.6.1 CK_RV

CK_RV is a value that identifies the return value of a Cryptoki function. It is defined as follows:

```
typedef CK_ULONG CK_RV;
```

The Cryptoki functions all return an error code or `CK_OK` to indicate success. The Cryptoki error codes are different from the Trusted Foundations's general error codes defined in section B1.2, "*Common Error codes*". They also use a different range of numbers (generic error codes use the range `0xFFFF0000`-`0xFFFFFFFF` and Cryptoki error codes are small numbers in the range `0x00000001`-`0x00000200`). Note, however, that the success value is zero in both cases.

See section B4.5.1, "*Function return values*" for a list of all supported error codes and their meaning.

B4.4 OBJECTS AND ATTRIBUTES

Cryptoki can recognize a number of classes of objects, as defined in the `CK_OBJECT_CLASS` data type. An object consists of a set of attributes, each of which has a given value. Each attribute that an object possesses has precisely one value.

The Cryptographic API supports the following object types:

- data objects (`CKO_DATA`)
- secret key objects (`CKO_SECRET_KEY`)
- private and public key objects (`CKO_PRIVATE_KEY` and `CKO_PUBLIC_KEY`).

The Cryptographic API provides functions for creating and destroying objects in general, and for obtaining the values of their attributes. Some of the cryptographic functions (e.g., `C_GenerateKey`) also create key objects to hold their results.

Objects cannot be modified once they are created. However, in some cases, objects can be copied using the function `C_CopyObject`.

Objects are always “well-formed” in the API—that is, an object always contains all required attributes, and the attributes are always consistent with one another from the time the object is created. This contrasts with some object-based paradigms where an object has no attributes other than perhaps a class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

Tables throughout this section and in the section B4.6, “*Cryptoki Mechanisms*” define each Cryptoki attribute in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly in section B4.3 (e.g., `CK_OBJECT_CLASS`). Attribute values may also take the following types:

- Byte array: an arbitrary string (array) of `CK_BYTE`s;
- Big integer: a string of `CK_BYTE`s representing an **unsigned** integer of arbitrary size, most-significant byte first (e.g., the integer 32768 is represented as the 2-byte string 0x80 0x00). Note that this representation differs from the signed integers used in ASN.1 encodings such as DER or BER.

In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of Cryptoki attributes. Some of these attributes possess default values, and need not be specified when creating an object. Nonetheless, the object possesses these attributes. A given object has a single value for each attribute it possesses.

B4.4.1 CREATING OBJECTS

All Cryptoki functions that create objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section B4.5.11) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see Section B4.6). In any case, all the required attributes supported by an object class that do not have default values must be specified when an object is created, either in the template or by the function itself.

Objects may be created with the Cryptoki functions `C_CreateObject`, `C_GenerateKey`, `C_GenerateKeyPair`, and `C_DeriveKey`.

Attempting to create an object with any of these functions requires an appropriate template to be supplied.

1. If the supplied template specifies a value for an invalid attribute, then the attempt fails with the error code `CKR_ATTRIBUTE_TYPE_INVALID`. An attribute is valid if it is mentioned in this specification;
2. If the supplied template specifies an invalid value for a valid attribute, then the attempt fails with the error code `CKR_ATTRIBUTE_VALUE_INVALID`. The valid values for Cryptoki attributes are described in this specification;
3. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are insufficient to

fully specify the object to create, then the attempt fails with the error code `CKR_TEMPLATE_INCOMPLETE`;

4. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are inconsistent, then the attempt fails with the error code `CKR_TEMPLATE_INCONSISTENT`. A set of attribute values is inconsistent if not all of its members can be satisfied simultaneously by the token, although each value individually is valid in Cryptoki. One example of an inconsistent template would be trying to create a secret key object with an attribute which is appropriate for various types of public keys or private keys, but not for secret keys. A final example would be a template with an attribute that violates some token specific requirement.

If more than one of the situations listed above applies to an attempt to create an object, then the error code returned from the attempt can be any of the error codes from above that applies.

In general, a Cryptoki function which requires a template for an object needs the template to specify—either explicitly or implicitly—any attributes that are not specified elsewhere. If a template specifies a particular attribute more than once, the function chooses the value of the last occurrence of the attribute in the template. A template being an array of attributes, the last occurrence means the one which has the highest index.

In this Cryptographic API, once an object is created, its attributes cannot be changed.

B4.4.2 COMMON ATTRIBUTES

This section defines the attributes that apply to all the types of objects supported in this specification. It defines the types of these attributes and provides details on when this attribute is required in templates depending on the function being called. It also defines whether the attribute can be retrieved using the function `C_GetAttributeValue`.

The following common attributes are defined:

Table B4-6: Cryptoki Common Attributes

Attribute Name:	**See section**
`CKA_CLASS`	B4.4.2.1
`CKA_TOKEN`	B4.4.2.2
`CKA_ID`	B4.4.2.3
`CKA_PRIVATE`	B4.4.2.4
`CKA_MODIFIABLE`	B4.4.2.5
`CKA_COPYABLE`	B4.4.2.6

B4.4.2.1 CKA_CLASS

Attribute Name:	`CKA_CLASS`
Type:	`CK_OBJECT_CLASS` with one of the following possible values:
Meaning:	This attribute is present on any object. It describes the type of object. It can have one of the following values: • `CKO_DATA` • `CKO_PUBLIC_KEY` • `CKO_PRIVATE_KEY` • `CKO_SECRET_KEY`
`C_CreateObject`:	Attribute must be provided in the template
`C_GenerateKey` / `C_GenerateKeyPair`:	Attribute is optional (what function is called unambiguously determines the object type)

`C_DeriveKey`:	The attribute must be provided in the template for the derived key
`C_GetAttributeValue`:	Attribute is always readable

B4.4.2.2 CKA_TOKEN

Attribute Name:	`CKA_TOKEN`
Type:	`CK_BBOOL`
Meaning:	This attribute is present on any object. It is a Boolean that determines whether the object is a token object (persistent) or a session object (temporary object attached to the session).
`C_CreateObject`, `C_DeriveKey`, `C_GenerateKey`:	Attribute is optional. If not set, a default value of `CK_FALSE` is assumed
`C_GenerateKeyPair`:	Attribute is optional. Default value is `CK_FALSE`. The public and private key must be of the same kind, either both token objects or both session objects. Otherwise, the function `C_GenerateKeyPair` returns `CKR_TEMPLATE_INCONSISTENT`
`C_GetAttributeValue`:	Attribute is always readable

B4.4.2.3 CKA_ID

Attribute Name:	`CKA_ID`
Type:	Byte array; can contain up to 64 bytes
Meaning:	This attribute is present on any token object. It must uniquely identify the object among all the token objects of its class in the token. In other words, within a given token, all the token objects are uniquely identified by the combination of their attributes `CKA_CLASS` and `CKA_ID`.
`C_CreateObject`, `C_DeriveKey`, `C_GenerateKey`, `C_GenerateKeyPair`	The attribute is mandatory when a token object is created, i.e., when the template contains the attribute `CKA_TOKEN` set to `CK_TRUE`. If you attempt to create a token object with the attribute `CKA_ID` set to an identifier already used by some existing object in the token with the same class (attribute `CKA_CLASS`), the function returns the error code `CKR_ATTRIBUTE_VALUE_INVALID`. When creating a session objects, you may specify a `CKA_ID` attribute in the template, but its value is ignored. Sessions objects have no identifiers. The only way to get a session object is to keep the handle returned by the function that created the object.
`C_GetAttributeValue`	For token objects, this attribute is always readable For session objects, this attribute never exists, even if it was specified when the object was created.

B4.4.2.4 CKA_PRIVATE

Attribute Name:	`CKA_PRIVATE`
Type:	Boolean
Meaning:	This attribute has a fixed value `CK_TRUE` because all objects are private in the Cryptographic API (see section B4.2.2, “Sessions”)

C_CreateObject, C_DeriveKey, C_GenerateKey, C_GenerateKeyPair:	You must not set this attribute or the function returns CKR_ATTRIBUTE_TYPE_INVALID
C_GetAttributeValue:	The attribute can be read and always has the value CK_TRUE.

B4.4.2.5 CKA_MODIFIABLE

Attribute Name:	CKA_MODIFIABLE
Type:	Boolean
Meaning:	This attribute has a fixed value CK_FALSE because no object can be modified (see section B4.2.2, “Sessions”)
C_CreateObject, C_DeriveKey, C_GenerateKey, C_GenerateKeyPair:	You must not set this attribute or the function returns CKR_ATTRIBUTE_TYPE_INVALID
C_GetAttributeValue:	The attribute can be read and always has the value CK_FALSE.

B4.4.2.6 CKA_COPYABLE

Attribute Name:	CKA_COPYABLE
Type:	CK_BBOOL
Meaning:	This attribute is present on any object. It is a Boolean that determines whether the object is copyable or not. See the specification of the function C_CopyObject in section B4.5.5.2 for more details.
C_CreateObject, C_CopyObject, C_DeriveKey, C_GenerateKey, C_GenerateKeyPair:	Attribute is optional. If not set when an object is created, the default value is CK_TRUE.
C_GetAttributeValue:	Attribute is always readable

B4.4.3 Data Objects Attributes

Data Objects (object class **CKO_DATA**) hold information defined by an application. Other than providing access to it, Cryptoki does not attach special meaning to the content of a data object.

The data objects can have the following attributes.

B4.4.3.1 CKA_VALUE (Data Object)

Attribute Name:	CKA_VALUE
Type:	Byte Array
Meaning:	Content of the data object. *Note that some products limit the size of the data object content. Consult the Product Reference Manual of your product.*
C_CreateObject	The value of the CKA_VALUE attribute must be provided. Once a data object is created, note that its content cannot be changed.

C_DeriveKey, C_GenerateKey, C_GenerateKeyPair:	*Not applicable to data objects*
C_GetAttributeValue:	The attribute can always be read.

B4.4.4 Key Objects Attributes

There is no CKO_ definition for the base key object class, only for the more tangible key types derived from it. These types are:

- CKO_PUBLIC_KEY,
- CKO_PRIVATE_KEY,
- CKO_SECRET_KEY.

All of these are valid object constant values for the type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of key object templates.

The following sections describe the general attributes that applies to key objects. The section B4.6, "*Cryptoki Mechanisms*" specifies the attributes that apply to particular types of keys.

The following attributes common to all types of keys are defined:

Table B4-7: Cryptoki Common Key Attributes

Attribute Name:	**See section**
CKA_KEY_TYPE	B4.4.4.1
CKA_SENSITIVE	B4.4.4.2
CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY, or CKA_DERIVE	B4.4.4.3
CKA_ALWAYS_AUTHENTICATE	B4.4.4.4
CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY	B4.4.4.5

B4.4.4.1 CKA_KEY_TYPE

Attribute Name:	CKA_KEY_TYPE
Type:	CK_KEY_TYPE
Meaning:	The type of the key. The following values are supported • CKK_AES: see section B4.6.4.1, "*AES secret key objects*" • CKK_DES: see section B4.6.5.1, "*DES secret key objects*" • CKK_DES2: see section B4.6.6.1, "*Double-Length DES Key Objects*" • CKK_DES3: see section B4.6.6.3, "*Triple-Length DES Key Objects*" • CKK_GENERIC_SECRET: see section B4.6.8.1, "*Generic Secret Key Objects*" • CKK_RC4: see section B4.6.7.1, "*RC4 secret key objects*" • CKK_RSA: see sections B4.6.1.1, "*RSA public key objects*" and B4.6.1.2, "*RSA private key objects*" • CKK_DSA: see sections B4.6.3.1, "*DSA public key objects*", and B4.6.3.2, "*DSA private key objects*" • CKK_DH: see sections B4.6.2.1, "*Diffie-Hellman public key objects*"

	and B4.6.2.2, "*Diffie-Hellman private key objects*" The object class must be consistent with the type of key: • secret keys (`CKK_AES`, `CKK_DES`, `CKK_DES2`, `CKK_DES3`, `CKK_GENERIC_SECRET`, `CKK_RC4`) must be of class `CKO_SECRET_KEY` • Public and private keys (`CKK_RSA`, `CKK_DSA`, and `CKK_DH`) must be of class `CKO_PRIVATE_KEY` or `CKO_PUBLIC_KEY`.
`C_CreateObject`	The value of the attribute must be provided when the key object is created
`C_GenerateKey`, `C_GenerateKeyPair`	The value of the attribute is optional. The generation mechanism always implicitly determine the type of key generated
`C_DeriveKey`	The value of the attribute is mandatory but only secret keys can be derived. See section B4.6.2.4, "*PKCS#3 Diffie-Hellman Key Derivation*" for more details
`C_GetAttributeValue`:	The attribute can always be read.

B4.4.4.2 CKA_SENSITIVE

Attribute Name:	`CKA_SENSITIVE`
Type:	Boolean
Meaning:	Whether the the attributes that constitute the key value can be retrieved with the function `C_GetAttributeValue`: • All public key objects are always non-sensitive and can be retrieved • By default, private key objects are sensitive and can never be retrieved in clear-text. However, if your product implements the Developer APIs version 3.1 or later, then a RSA private key can be marked as non-sensitive when imported or generated so that the private value later be retrieved in clear-text. • In general, secret key objects are sensitive. However, when a secret key object is derived, it can be marked as non-sensitive subject to some limitations. See section B4.6.2.4, "*PKCS#3 Diffie-Hellman Key Derivation*" for more details
`C_CreateObject`, `C_GenerateKey`, `C_GenerateKeyPair`	The value of the attribute must not be set. A default value is always determined by the implementation: • `CK_FALSE` for public key objects • `CK_TRUE` for private and secret key objects As an exception, if your product implements the Developer APIs V3.1 or later, it is possible to specify a value for the attribute `CKA_SENSITIVE`, but only for RSA private keys, either when importing a RSA private key using the function `C_CreateObject` or when generating a RSA key-pair using the function `C_GenerateKeyPair`. The value of the attribute can be set to `CK_FALSE`, which allows the protected attributes of a RSA private key to be later retrieved. If the attribute is set to `CK_FALSE`, or if it is not specified at all, then the protected attributes cannot be retrieved.
`C_DeriveKey`	The attribute is optional. When not specified, a default value of `CK_TRUE` is assumed. The attribute can be set to `CK_FALSE` but only if the base key has the attribute `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY` set to `CK_TRUE`.

C_GetAttributeValue:	The attribute can always be read.

B4.4.4.3 CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY, CKA_DERIVE

Attribute Name:	CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, CKA_VERIFY, or CKA_DERIVE
Type:	Boolean
Meaning:	This set of attributes are used to control whether the key object can be used with certain kinds of operations: • CKA_ENCRYPT controls whether the object can be used with the function C_EncryptInit • CKA_DECRYPT controls whether the object can be used with the function C_DecryptInit • CKA_SIGN controls whether the object can be used with the function C_SignInit • CKA_VERIFY controls whether the object can be used with the function C_VerifyInit • CKA_VERIFY controls whether the object can be used with the function C_VerifyInit • CKA_DERIVE controls whether the object can be used with the function C_DeriveInit If the value is set to CK_TRUE, the operation is allowed. Otherwise, the operation is forbidden and the function returns CKR_KEY_FUNCTION_NOT_PERMITTED.
C_CreateObject, C_GenerateKey, C_GenerateKeyPair, C_DeriveKey	The value of these attribute is optional. When not specified, a default value of CK_FALSE (operation not allowed) is assumed. Not all of these attributes apply to all key types: • CKA_ENCRYPT applies to all secret keys and RSA public keys • CKA_DECRYPT applies to all secret keys and RSA private keys • CKA_SIGN applies to RSA and DSA private keys and to generic secret keys (for HMAC) • CKA_VERIFY applies to RSA and DSA public keys and to generic secret keys (for HMAC) • CKA_DERIVE: applies only to Diffie-Hellman private keys
C_GetAttributeValue:	These attribute can always be read.

B4.4.4.4 CKA_ALWAYS_AUTHENTICATE

Attribute Name:	CKA_ALWAYS_AUTHENTICATE
Type:	Boolean
Meaning:	Fixed value CK_FALSE. This attribute applies only to private key objects and means that there is no need to login to use the key
C_CreateObject, C_GenerateKey, C_GenerateKeyPair, C_DeriveKey	The value of these attribute must not be set. A default value of CK_FALSE is always assumed
C_GetAttributeValue:	If the key is a CKO_PRIVATE_KEY, the attribute can be read and always

	has the value `CK_FALSE`. Otherwise, the attribute does not exist.

B4.4.4.5 CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY

Attribute Name:	`CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`
Type:	Boolean
Meaning:	Non-standard attribute. When set to `CK_TRUE` on a base key, it allows keys derived from the base key to be marked as non-sensitive. Otherwise, derived keys are necessarily sensitive. Nonetheless, the attribute can be set and retrieved on all key types.
`C_CreateObject`, `C_GenerateKey`, `C_GenerateKeyPair`, `C_DeriveKey`	Optional. Default value is `CK_FALSE`.
`C_GetAttributeValue`:	Attribute always readable.

B4.4.4.6 OTHER KEY OBJECTS ATTRIBUTES

This section provides a summary of all the other attributes that can be set for key objects. These attributes do not apply to all key types. They are described more precisely in the section B4.6, "*Cryptoki Mechanisms"* where each type of key is specified.

The key objects attribute are either non-sensitive or protected. Non-sensitive attributes can always be retrieved with the function `C_GetAttributeValue`. Protected attributes belong to secret or private keys and cannot be retrieved except if the key is marked as non-sensitive, i.e., with the attribute `CKA_SENSITIVE` set to `CK_FALSE`.

The non-senstive attributes are described in the following table:

Table B4-8: Cryptoki Non-Sensitive Key Attributes

Attribute Name	Key types	Attribute Type	Meaning
`CKA_VALUE_LEN`	all secret keys	`CK_ULONG`	Length of the secret key in bytes
`CKA_MODULUS`	RSA public and private keys	Big integer	Modulus *n*
`CKA_PUBLIC_EXPONENT`		Big integer	Public exponent *e*. When not specified, a default value of 0x0100001 is assumed.
`CKA_MODULUS_BITS`	RSA public keys	`CK_ULONG`	Length in bits of the modulus
`CKA_VALUE` (public)	DSA and DH public keys	Big integer	Public value *y*. Note that the attribute CKA_VALUE, when applied to a secret or private key is a protected attribute.
`CKA_PRIME`	DSA and DH public and private keys	Big integer	Prime *p*
`CKA_BASE`		Big integer	Generator *g*
`CKA_SUBPRIME`	DSA public and private keys	Big integer	Subprime *q*

Attribute Name	Key types	Attribute Type	Meaning
`CKA_VALUE_BITS`	DH private keys	`CK_ULONG`	Length of the DH private key *x* in bits

The protected attributes are described in the following table:

Table B4-9: Cryptoki Protected Key Attributes

Attribute Name	Key types	Attribute Type	Meaning
`CKA_VALUE` (secret)	All secret keys	Byte array	The content of the secret key.
`CKA_VALUE` (private)	DSA and DH private keys	Big integer	Private value *x*
`CKA_PRIVATE_EXPONENT`	RSA private keys	Big integer	Private exponent *d*
`CKA_PRIME_1`		Big integer	Prime *p*
`CKA_PRIME_2`		Big integer	Prime *q*
`CKA_EXPONENT_1`		Big integer	Private exponent *d* module *p-1*
`CKA_EXPONENT_2`		Big integer	Private exponent *d* module *q-1*
`CKA_COEFFICIENT`		Big integer	CRT coefficient q^{-1} mod *p*

B4.5 FUNCTIONS

This section gives details of the functions of the API, and the various error codes that these functions may return.

- If a Cryptoki function executes successfully, it returns the value `CKR_OK`.
- If a Cryptoki function does not execute successfully, it returns some value other than `CKR_OK`, and the token is in the same state as it was in prior to the function call. If the function call was supposed to modify the contents of certain memory addresses on the host computer, these memory addresses may have been modified, despite the failure of the function.
- In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value `CKR_GENERAL_ERROR`. When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

There are a small number of Cryptoki functions whose return values do not behave precisely as described above; these exceptions are documented individually with the description of the functions themselves.

B4.5.1 FUNCTION RETURN VALUES

The Cryptoki interface possesses a number of functions and return values. The Table B4-10 enumerates the various possible return values for Cryptoki functions.

Note that the Cryptoki error codes are different from the Trusted Foundations's general error codes defined in section B1.2, "*Common Error codes*". They also use a different range of numbers (generic error codes use the range `0xFFFF0000`-`0xFFFFFFFF` and Cryptoki error codes are small numbers in the range `0x00000001`-`0x00000200`). Note, however, that the success value is zero in both cases.

Table B4-10: Cryptoki Error Codes

PKCS#11 Error Codes	Value
CKR_OK	0x00000000
CKR_HOST_MEMORY	0x00000002
CKR_SLOT_ID_INVALID	0x00000003
CKR_GENERAL_ERROR	0x00000005
CKR_ARGUMENTS_BAD	0x00000007
CKR_ATTRIBUTE_SENSITIVE	0x00000011
CKR_ATTRIBUTE_TYPE_INVALID	0x00000012
CKR_ATTRIBUTE_VALUE_INVALID	0x00000013
CKR_DATA_INVALID	0x00000020
CKR_DATA_LEN_RANGE	0x00000021
CKR_DEVICE_ERROR	0x00000030
CKR_DEVICE_MEMORY	0x00000031
CKR_ENCRYPTED_DATA_INVALID	0x00000040
CKR_ENCRYPTED_DATA_LEN_RANGE	0x00000041
CKR_KEY_HANDLE_INVALID	0x00000060
CKR_KEY_SIZE_RANGE	0x00000062
CKR_KEY_TYPE_INCONSISTENT	0x00000063
CKR_KEY_FUNCTION_NOT_PERMITTED	0x00000068
CKR_MECHANISM_INVALID	0x00000070
CKR_MECHANISM_PARAM_INVALID	0x00000071
CKR_OBJECT_HANDLE_INVALID	0x00000082
CKR_OPERATION_ACTIVE	0x00000090
CKR_OPERATION_NOT_INITIALIZED	0x00000091
CKR_SESSION_HANDLE_INVALID	0x000000B3
CKR_SESSION_READ_ONLY	0x000000B5
CKR_SESSION_PARALLEL_NOT_SUPPORTED	0x000000B4
CKR_SIGNATURE_INVALID	0x000000C0

PKCS#11 Error Codes	Value
CKR_SIGNATURE_LEN_RANGE	0x000000C1
CKR_TEMPLATE_INCOMPLETE	0x000000D0
CKR_TEMPLATE_INCONSISTENT	0x000000D1
CKR_TOKEN_NOT_PRESENT	0x000000E0
CKR_RANDOM_SEED_NOT_SUPPORTED	0x00000120
CKR_RANDOM_NO_RNG	0x00000121
CKR_DOMAIN_PARAMS_INVALID	0x00000130
CKR_BUFFER_TOO_SMALL	0x00000150
CKR_CRYPTOKI_NOT_INITIALIZED	0x00000190
CKR_CRYPTOKI_ALREADY_INITIALIZED	0x00000191

B4.5.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

- CKR_GENERAL_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.
- CKR_HOST_MEMORY: The process that calls the Cryptographic API has insufficient memory to perform the requested function.
- CKR_OK: The function executed successfully.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_GENERAL_ERROR or CKR_HOST_MEMORY would be an appropriate error return, then CKR_GENERAL_ERROR should be returned.

B4.5.1.2 Cryptoki function return values for functions that use a session handle

Any Cryptoki function that takes a session handle as one of its arguments (*i.e.*, any Cryptoki function except for C_Initialize, C_Finalize, C_GetInfo, and C_OpenSession) can return: CKR_SESSION_HANDLE_INVALID: The specified session handle was invalid *at the time that the function was invoked*.

B4.5.1.3 Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for C_Initialize, C_Finalize or C_GetInfo) can return any of the following values:

- CKR_DEVICE_MEMORY: The token does not have sufficient memory to perform the requested function.
- CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot.
- CKR_TOKEN_NOT_PRESENT: The token was not present in its slot at the time a session is open or the application authentication fails. For the **External Cryptographic API**, this means that the authentication of the calling application fails as described in section B4.2.2. For the **Internal Cryptographic API**, this error cannot be returned.

B4.5.1.4 All other Cryptoki function return values

Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions of particular error codes, there are in general no particular priorities among the errors listed below, *i.e.*, if more than one error code might apply to an execution of a function, then the function may return any applicable error code.

- CKR_ARGUMENTS_BAD: This is a rather generic error code which indicates that the arguments supplied to the Cryptoki function were in some way not appropriate. This error is set when an error is detected on the function parameters and when no other error code returned by the function is appropriate. For example, it is returned when a pointer parameter is NULL and the function does not permit it.

- `CKR_ATTRIBUTE_SENSITIVE`: An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or unextractable.
- `CKR_ATTRIBUTE_TYPE_INVALID`: An invalid attribute type was specified in a template. See section B4.4.1 for more information.
- `CKR_ATTRIBUTE_VALUE_INVALID`: An invalid value was specified for a particular attribute in a template. See section B4.4.1 for more information.
- `CKR_BUFFER_TOO_SMALL`: The output of the function is too large to fit in the supplied buffer.
- `CKR_CRYPTOKI_ALREADY_INITIALIZED`: This value can only be returned by `C_Initialize`. It means that the Cryptoki library has already been initialized (by a previous call to `C_Initialize` which did not have a matching `C_Finalize` call).
- `CKR_CRYPTOKI_NOT_INITIALIZED`: This value can be returned by any function other than C_Initialize. It indicates that the function cannot be executed because the Cryptoki library has not yet been initialized by a call to C_Initialize.
- `CKR_DATA_INVALID`: The plaintext input data to a cryptographic operation is invalid (e.g bad parity bits in an input DES key). This return value has lower priority than `CKR_DATA_LEN_RANGE`.
- `CKR_DATA_LEN_RANGE`: The plaintext input data to a cryptographic operation has a bad length. Depending on the operation's mechanism, this could mean that the plaintext data is too short, too long, or is not a multiple of some particular blocksize. This return value has higher priority than `CKR_DATA_INVALID`.
- `CKR_DOMAIN_PARAMS_INVALID`: Invalid or unsupported domain parameters were supplied to the function.
- `CKR_ENCRYPTED_DATA_INVALID`: The encrypted input to a decryption operation has been determined to be invalid ciphertext (e.g bad padding). This return value has lower priority than `CKR_ENCRYPTED_DATA_LEN_RANGE`.
- `CKR_ENCRYPTED_DATA_LEN_RANGE`: The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular blocksize. This return value has higher priority than `CKR_ENCRYPTED_DATA_INVALID`.
- `CKR_KEY_FUNCTION_NOT_PERMITTED`: An attempt has been made to use a key for a cryptographic purpose that the key's attributes are not set to allow it to do. For example, to use a key for performing encryption, that key must have its `CKA_ENCRYPT` attribute set to `CK_TRUE` (the fact that the key must have a `CKA_ENCRYPT` attribute implies that the key cannot be a private key). This return value has lower priority than `CKR_KEY_TYPE_INCONSISTENT`.
- `CKR_KEY_HANDLE_INVALID`: The specified key handle is not valid. We reiterate here that 0 is never a valid key handle.
- `CKR_KEY_SIZE_RANGE`: Although the requested keyed cryptographic operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the range of key sizes that it can handle.
- `CKR_KEY_TYPE_INCONSISTENT`: The specified key is not the correct type of key to use with the specified mechanism. This return value has a higher priority than `CKR_KEY_FUNCTION_NOT_PERMITTED`.
- `CKR_MECHANISM_INVALID`: An invalid mechanism was specified to the cryptographic operation. This error code is an appropriate return value if an unknown mechanism was specified or if the mechanism specified cannot be used in the selected token with the selected function.
- `CKR_MECHANISM_PARAM_INVALID`: Invalid parameters were supplied to the mechanism specified to the cryptographic operation. Which parameter values are supported by a given mechanism can vary from token to token.
- `CKR_OBJECT_HANDLE_INVALID`: The specified object handle is not valid. We reiterate here that 0 is never a valid object handle.
- `CKR_OPERATION_ACTIVE`: There is already an active operation which prevents Cryptoki from activating the specified operation. For example, an active object-searching operation would

prevent Cryptoki from activating an encryption operation with C_EncryptInit. See section B4.2.2 for more details.

- CKR_OPERATION_NOT_INITIALIZED: There is no active operation of an appropriate type in the specified session. For example, an application cannot call C_Encrypt in a session without having called C_EncryptInit first to activate an encryption operation.
- CKR_RANDOM_NO_RNG: This value can be returned by C_SeedRandom and C_GenerateRandom. It indicates that the specified token doesn't have a random number generator. This return value has higher priority than CKR_RANDOM_SEED_NOT_SUPPORTED.
- CKR_RANDOM_SEED_NOT_SUPPORTED: This value can only be returned by C_SeedRandom. It indicates that the token's random number generator does not accept seeding from an application. This return value has lower priority than CKR_RANDOM_NO_RNG.
- CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, no token supports parallel sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by C_OpenSession, and it is only returned when C_OpenSession is called in a particular deprecated way.
- CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action because it is a read-only session.
- CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid solely on the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.
- CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower priority than CKR_SIGNATURE_LEN_RANGE.
- CKR_SLOT_ID_INVALID: The specified slot ID is not valid.
- CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and lacks some necessary attributes. See section B4.4.1 for more information.
- CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting attributes. See section B4.4.1 for more information.

B4.5.1.5 MORE ON RELATIVE PRIORITIES OF CRYPTOKI ERRORS

In general, when a Cryptoki call is made, error codes from Section B4.5.1.1 (other than CKR_OK) take precedence over error codes from Section B4.5.1.2, which take precedence over error codes from Section B4.5.1.3, which take precedence over error codes from Section B4.5.1.4. Other than these precedences, if more than one error code applies to the result of a Cryptoki call, any of the applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the descriptions of functions.

B4.5.1.6 ERROR CODE "GOTCHAS"

Here is a short list of a few particular things about return values that Cryptoki developers might want to be aware of:

- The difference between CKR_DATA_INVALID and CKR_DATA_LEN_RANGE can be somewhat subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to always treat them equivalently.
- Similarly, the difference between CKR_ENCRYPTED_DATA_INVALID and CKR_ENCRYPTED_DATA_LEN_RANGE can be subtle, and it may be best to treat these return values equivalently.

B4.5.2 CONVENTIONS FOR FUNCTIONS RETURNING OUTPUT IN A VARIABLE-LENGTH BUFFER

A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism. The amount of output returned by these functions is returned in a variable-length application-supplied buffer. An example of a function of this sort is C_Encrypt, which takes some plaintext as an argument, and outputs a buffer full of ciphertext.

These functions have some common calling conventions, which are described here. Two of the arguments to the function are a pointer to the output buffer (say `pBuf`) and a pointer to a location which will hold the length of the output produced (say `pulBufLen`). There are two ways for an application to call such a function:

- If `pBuf` is `NULL`, then all that the function does is return (in `*pulBufLen`) a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may somewhat exceed the precise number of bytes needed, but should not exceed it by a large amount. `CKR_OK` is returned by the function.
- If `pBuf` is not `NULL`, then `*pulBufLen` must contain the size in bytes of the buffer pointed to by `pBuf`. If that buffer is large enough to hold the cryptographic output produced from the input to the function, then that cryptographic output is placed there, and `CKR_OK` is returned by the function. If the buffer is not large enough, then `CKR_BUFFER_TOO_SMALL` is returned. In either case, `*pulBufLen` is set to hold the *exact* number of bytes needed to hold the cryptographic output produced from the input to the function.

All functions which use the above convention will explicitly say so.

B4.5.3 General-Purpose Functions

Cryptoki provides the following general-purpose functions:

B4.5.3.1 C_Initialize

```
CK_RV C_Initialize(
           void* pInitArgs
);
```

`C_Initialize` initializes the Cryptoki library.

`C_Initialize` should be the first Cryptoki call made by an application. What this function actually does is implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or any other resources it requires.

If several applications are using Cryptoki, each one should call `C_Initialize`. Every call to `C_Initialize` should (eventually) be succeeded by a single call to `C_Finalize`.

In the **Internal Cryptographic API**, the Cryptographic API is always implicitly initialized. The function `C_Initialize` does nothing and calling it is optional.

In the **External Cryptographic API**, the parameter `pInitArgs` is reserved for future use and must be set to `NULL`, otherwise the function returns `CKR_ARGUMENTS_BAD`.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_ALREADY_INITIALIZED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`.

Example: see C_GetInfo.

B4.5.3.2 C_FINALIZE

```
CK_RV C_Finalize(
           void* pReserved
);
```

C_Finalize is called to indicate that an application is finished with the Cryptoki library. It should be the last Cryptoki call made by an application. The pReserved parameter is reserved for future versions; for this version, it should be set to NULL (if C_Finalize is called with a non-NULL value for pReserved, it should return the value CKR_ARGUMENTS_BAD).

If several applications are using Cryptoki, each one should call C_Finalize. Each application's call to C_Finalize should be preceded by a single call to C_Initialize; in between the two calls, an application can make calls to other Cryptoki functions.

In the **Internal Cryptographic API**, the Cryptographic API is always implicitly initialized. The function C_Finalize does nothing and calling it is optional.

In the **External Cryptographic API**, the parameter pReserved is reserved for future use and must be set to NULL, otherwise the function returns CKR_ARGUMENTS_BAD.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

Example: see C_GetInfo.

B4.5.3.3 C_GETINFO

```
CK_RV C_GetInfo(
            CK_INFO* pInfo
);
```

`C_GetInfo` returns general information about Cryptoki. `pInfo` points to the location that receives the information.

In Trusted Foundations' Cryptographic API, the function fills `pInfo` with the following information:

Field Name	Value
cryptokiVersion	{ 2, 20} // Spec 2.20
manufacturerID	"Trusted Logic "
flags	0
libraryDescription	"PKCS#11 "
libraryVersion	{ 3, 0 } // v 3.0

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_OK`.

Example:

```
CK_INFO info;
CK_RV rv;

rv = C_Initialize(NULL);
assert(rv == CKR_OK);

rv = C_GetInfo(&info);
assert(rv == CKR_OK);

if(info.cryptokiVersion.major == 2) {
  /* Perform some useful operations using Cryptoki */
  ...
}

rv = C_Finalize(NULL);
assert(rv == CKR_OK);
```

B4.5.4 SESSION MANAGEMENT FUNCTIONS

A typical application might perform the following series of steps to make use of a token (note that there are other reasonable sequences of events that an application might perform):

1. Select a token.
2. Make one call to `C_OpenSession` to obtain one session with the token.
3. Perform cryptographic operations using the sessions with the token.
4. Call `C_CloseSession`.
5. Note that it is not necessary to call `C_Login` because the session is implicitly in the "User Functions" state as soon as it is opened.

Cryptoki provides the following functions for session management:

B4.5.4.1 C_OPENSESSION

```
CK_RV C_OpenSession (
           CK_SLOT_ID            slotID,
           CK_FLAGS              flags,
           void*                 pApplication,
           CK_NOTIFY             Notify,
           CK_SESSION_HANDLE*    phSession
);
```

`C_OpenSession` can be used:

- to open a new primary session between an application or service and a token in a particular slot*;*
- or to open a new secondary session within a given primary session.

The two cases are distinguished by the presence of the non-standard flag `CKVF_OPEN_SUB_SESSION`:

- If `flags` does not contain the flag `CKVF_OPEN_SUB_SESSION`, then a new primary session is opened. In this case, `slotID` describes the the slode identifier and `phSession` points to the location that receives the handle for the new session
- If `flags` contains the flag `CKVF_OPEN_SUB_SESSION`, then a new secondary session is opened. In this case, `phSession` must point to an existing handle on an opened primary session. When the function completes, `phSession` is filled with a handle on the newly-created secondary session. The content of `slotID` is ignored.

Opening a Primary Session

When opening a primary session, the `flags` parameter consists of the logical OR of zero or more bit flags defined in the `CK_SESSION_INFO` data type (see section B4.3.3.3.3). This parameter can only take one of the following two values:

- `CKF_SERIAL_SESSION`: opens a Read-Only session: in such a session, no token object can be created or deleted
- `CKF_SERIAL_SESSION | CKF_RW_SESSION`: opens a Read-Write session, in which token objects can be created or deleted

Note that there is no limitation on the number of Read-Only or Read-Write sessions that can be simultaneously opened on a token, except the system resources.

For legacy reasons, the `CKF_SERIAL_SESSION` bit must always be set; if a call to `C_OpenSession` does not have this bit set, the call returns unsuccessfully with the error code `CKR_SESSION_PARALLEL_NOT_SUPPORTED`.

The slot identifier gives access to a virtual token that can depend on the identity of the caller and other parameters as described in section B4.2.1, "*Slot Identifiers, Virtual Tokens, and Token Storages*".

The virtual tokens accessible to callers of `C_OpenSession` depend on the calling world and the slot identifier (parameter `slotID`). The Table B4-11 lists the slot identifiers for the Normal World and the Table B4-12 lists the slot identifiers for the Secure World.

Table B4-11: Virtual Tokens Accessible from the Normal World

Slot Identifier	**Corresponding Virtual Token**

`0`
`(CKV_TOKEN_SYSTEM_SHARED)`
	A virtual token shared between all normal world applications and stored in the system storage area. Objects stored in this token by one application can be seen and used by all other normal world applications

`0x4012`
`(CKV_TOKEN_USER_SHARED)`
	A virtual token shared between all normal world applications and stored in the user storage area

`1`
`(CKV_TOKEN_SYSTEM)`
	A virtual token private to the normal world application, created on demand, and stored in the system storage area.
	Different applications have access to different application-private token, even if they both use the slot identifier 1. How applications are differentiated relies on the TF Client API following login types:
	• If the function `TEEC_ReadSignatureFile` returns a signature file for the calling application, then the login type `TEEC_LOGIN_AUTHENTICATION` is used, which means that applications are distinguished by the UUID stored in their signature files
	• If no signature file is found, then the login type `TEEC_LOGIN_USER_APPLICATION` is used, which means that the separation of applications depends on the Operating System. It typically involves elements that are related to the application itself but also to the user executing the application in the case of a multi-user OS.
	See section B3.3.5.6 for more details on these login types.

`0x4004`
`(CKV_TOKEN_USER)`
	A virtual token private to the application, created on demand, and stored in the user storage area. The applications are separated is the same as for `CKV_TOKEN_SYSTEM`.

`0x0001gggg`
`(CKV_TOKEN_SYSTEM_GROUP(0xgggg))`
	A virtual token shared between all applications that belong that the group `0xgggg` (between 0 and 65535) and stored in the system storage area.
	The system is responsible to check that the calling application does belong to specified group and to prevent an application from impersonating group membership. This makes use of the TF Client API login type `TEEC_LOGIN_GROUP`. See section B3.3.5.6 for mode details on this login type.

`0x0002gggg`
`(CKV_TOKEN_USER_GROUP(0xgggg))`
	Same as `CKV_TOKEN_SYSTEM_GROUP(0xgggg)`, but stored in the user storage area.

Table B4-12: Virtual Tokens Accessible from the Secure World

Slot Identifier	**Corresponding Virtual Token**

1 (S_CRYPTOKI_KEYSTORE_PRIVATE)	Virtual token private to the service, created on demand, and stored in the system storage area. The token is associated with the service UUID. Different services have therefore access to different virtual tokens
0x4004 (S_CRYPTOKI_KEYSTORE_PRIVATE_USER)	Same as S_CRYPTOKI_KEYSTORE_PRIVATE but stored in the user storage area.

Note that virtual tokens for the Normal World and for the Secure World are always separated.

Opening a Secondary Session

Opending a secondary session is expressed by setting the flag CKVF_OPEN_SUB_SESSION in the parameter flags. In this case, the parameter slotID is ignored and, before calling this function, *phSession must be set to a handle on an existing primary session.

A secondary session can be used to start new stateful operations independently of the operations in the primary session or in any of the secondary sessions opened within the primary session. A secondary session does not provide a scope for object handles: if any object handle is created or deleted within a secondary session, the effect is visible in the parent primary session and in all secondary sessions attached to the primary session. When the secondary session is closed, it has no effect on any object handles.

The Cryptographic API does not support callbacks. The parameter pApplication and Notify are always ignored.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_PARALLEL_NOT_SUPPORTED, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

B4.5.4.2 C_CLOSESESSION

```
CK_RV C_CloseSession(
            CK_SESSION_HANDLE hSession
);
```

`C_CloseSession` closes a primary session between an application and a token or a secondary session opened within a primary session.

- If `hSession` refers to a primary session handle, all session objects created in the session are automatically destroyed and all secondary sessions opened within the primary sessions are automatically closed
- If `hSession` refers to a secondary session handle, `C_CloseSession` has no effect on the object handles, it just releases the resource associated with the secondary session. If an operation is pending within the secondary session, it is terminated.

Internal API Notes

- the function `SHandleClose` function can also be used to close a cryptoki session. When called on a handle of Cryptoki session, it is a synonym of `C_CloseSession`.
- If a Cryptoki Update Shortcut is active when the session is closed, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
CK_SLOT_ID slotID;
CK_SESSION_HANDLE hSession;
CK_RV rv;
...

rv = C_OpenSession(1,
                   CKF_RW_SESSION | CKF_SERIAL_SESSION,
                   NULL,
                   NULL,
                   &hSession);

/* Perform some useful operations using Cryptoki */

if (rv == CKR_OK) {
  ...
  C_CloseSession(hSession);
}
```

B4.5.4.3 C_LOGIN

```
CK_RV C_Login(
           CK_SESSION_HANDLE hSession,
           CK_USER_TYPE      userType,
           CK_UTF8CHAR*      pPin,
           CK_ULONG          ulPinLen
);
```

This function is kept for backward-compatibility but does nothing. The function always returns `CKR_OK`.

The user is assumed to be automatically authenticated when s/he opens a session.

Return values: `CKR_OK`.

B4.5.4.4 C_LOGOUT

```
CK_RV C_Login(
           CK_SESSION_HANDLE hSession
);
```

This function is kept for backward-compatibility but does nothing. The function always returns `CKR_OK`.

Return values: `CKR_OK`.

B4.5.5 Object Management Functions

Cryptoki provides the following functions for managing objects. Additional functions provided specifically for managing key objects are described in Section B4.5.11.

B4.5.5.1 C_CreateObject

```
CK_RV C_CreateObject(
            CK_SESSION_HANDLE    hSession,
            CK_ATTRIBUTE*        pTemplate,
            CK_ULONG             ulCount,
            CK_OBJECT_HANDLE*    phObject
);
```

`C_CreateObject` creates a new object. `hSession` is the session's handle; `pTemplate` points to the object's template; `ulCount` is the number of attributes in the template; `phObject` points to the location that receives the new object's handle.

If a call to `C_CreateObject` cannot support the precise template supplied to it, it will fail and return without creating any object.

Only session objects can be created during a read-only session. If an attempt is made to create a token object in a read-only session, this function returns `CKR_SESSION_READ_ONLY`.

The expected attributes depend on the object type being created. Refer to the supported attribute list in section B4.4, "*Objects and Attributes*" for details of what attributes are allowed, and what values they can take.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_ATTRIBUTE_TYPE_INVALID`, `CKR_ATTRIBUTE_VALUE_INVALID`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SESSION_READ_ONLY`, `CKR_TEMPLATE_INCOMPLETE`, `CKR_TEMPLATE_INCONSISTENT`.

Example:

```
CK_RV              rv;
CK_SESSION_HANDLE  hSession;
CK_OBJECT_CLASS    keyClass   = CKO_PUBLIC_KEY;
CK_KEY_TYPE        keyType   = CKK_RSA;
CK_BYTE            objId[2]  = {0x00, 0x01};
CK_BBOOL           bTrue = TRUE;
CK_BBOOL           bFalse = FALSE;
CK_OBJECT_HANDLE   hKey;
CK_BYTE modulus[]  = {...};
CK_BYTE exponent[] = {...};


 CK_ATTRIBUTE publicKeyTemplate[7] =
 {
   {CKA_CLASS,           &keyClass,  sizeof(keyClass) },
   {CKA_ID,              objId,      sizeof(objId)    },
   {CKA_TOKEN,           &bTrue,     sizeof(bTrue)    },
   {CKA_KEY_TYPE,        &keyType,   sizeof(keyType)  },
   {CKA_ENCRYPT,         &bTrue,     sizeof(bTrue)    },
   {CKA_MODULUS,         modulus,    sizeof(modulus)  },
   {CKA_PUBLIC_EXPONENT, exponent,   sizeof(exponent) }
 };

 rv = C_CreateObject(hSession,
                     publicKeyTemplate,
                     7,
                     &hKey);
 if (rv == CKR_OK) {
   ...
```

```
    C_DestroyObject(hSession, hKey);
  }
```

B4.5.5.2 C_COPYOBJECT

```
CK_RV C_CopyObject(
            CK_SESSION_HANDLE    hSession,
            CK_OBJECT_HANDLE     hObject
            CK_ATTRIBUTE*        pTemplate,
            CK_ULONG             ulCount,
            CK_OBJECT_HANDLE*    phNewObject
);
```

`C_CopyObject` copies an object, creating a new object for the copy. `hSession` is the session's handle; `hObject` is the object's handle; `pTemplate` points to the template for the new object; `ulCount` is the number of attributes in the template; `phNewObject` points to the location that receives the handle for the copy of the object.

The template must specify the new values for the attributes of the object that can be "modified", according to the table Table B4-13(no other attributes is allowed). If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code `CKR_TEMPLATE_INCONSISTENT`.

Table B4-13: Attributes modifiable from initial object to new object

Attribute Name:	**Restriction:**
`CKA_TOKEN`	The target object must be a *token* object. The attribute `CKA_TOKEN` is: • Mandatory if the source key is a session object. In that case, the copy template MUST contain `CKA_TOKEN` and it must be set to `CK_TRUE`. • Optional if the source key is a token object. In that case, if `CKA_TOKEN` is present in the template, it must be set to `CK_TRUE`.
`CKA_ID`	Mandatory. Can take any content (0 to 64 bytes) provided it does not correspond to an existing object with the same class. In other words, copying an object must not violate the constraint that a token cannot contain two objects with the same class (attribute `CKA_CLASS`) and the same identifier (attribute `CKA_ID`). Otherwise, the call fails with the error `CKR_ATTRIBUTE_VALUE_INVALID`.

If a call to `C_CopyObject` cannot support the precise template supplied to it, it will fail and return without creating any object. If the object indicated by `hObject` has its `CKA_COPYABLE` attribute set to `CK_FALSE`, `C_CopyObject` will return `CKR_COPY_PROHIBITED`.

Only session objects can be created during a read-only session. If an attempt is made to create a token object in a read-only session, this function returns `CKR_SESSION_READ_ONLY`.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_ATTRIBUTE_TYPE_INVALID`, `CKR_ATTRIBUTE_VALUE_INVALID`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SESSION_READ_ONLY`, `CKR_TEMPLATE_INCONSISTENT`, `CKR_COPY_PROHIBITED`.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hKey, hNewKey;
CK_OBJECT_CLASS   keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE       keyType = CKK_DES;
CK_BYTE           id[2] = {0x00, 0x01};
CK_BYTE           keyValue[] = {...};
CK_BBOOL          false = CK_FALSE;
CK_BBOOL          true  = CK_TRUE;

CK_ATTRIBUTE keyTemplate[] = {
   {CKA_CLASS,    &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType,  sizeof(keyType)},
   {CKA_TOKEN,    &false,    sizeof(false)},
   {CKA_VALUE,    keyValue,  sizeof(keyValue)}
};
CK_ATTRIBUTE copyTemplate[] = {
   {CKA_TOKEN,    &true,     sizeof(true)},
   {CKA_ID,       id,        sizeof(id)},
};
CK_RV rv;
.
.
/* Create a DES secret key session object */
rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
if (rv == CKR_OK) {
   /* Create a copy which is a token object */
   rv = C_CopyObject(hSession, hKey, &copyTemplate, 1, &hNewKey);
   ...
}
```

B4.5.5.3 C_DestroyObject

```
CK_RV C_DestroyObject(
           CK_SESSION_HANDLE hSession,
           CK_OBJECT_HANDLE  hObject
);
```

`C_DestroyObject` destroys an object. `hSession` is the session's handle and `hObject` is the object's handle.

Only session objects can be destroyed during a read-only session. If an attempt is made to destroy a token object in a read-only session, this function returns `CKR_SESSION_READ_ONLY`.

Return values: `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OBJECT_HANDLE_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SESSION_READ_ONLY`.

Example: see C_CreateObject.

B4.5.5.4 C_GetAttributeValue

```
CK_RV C_GetAttributeValue(
           CK_SESSION_HANDLE hSession,
           CK_OBJECT_HANDLE  hObject,
           CK_ATTRIBUTE*     pTemplate,
           CK_ULONG          ulCount
);
```

C_GetAttributeValue obtains the value of one or more attributes of an object. hSession is the session's handle; hObject is the object's handle; pTemplate points to a template that specifies which attribute values are to be obtained, and receives the attribute values; ulCount is the number of attributes in the template.

For each (type, pValue, ulValueLen) triple in the template, C_GetAttributeValue performs the following algorithm:

1. If the specified attribute (*i.e.*, the attribute specified by the type field) for the object cannot be revealed because the object is sensitive, then the ulValueLen field in that triple is modified to hold the value -1 (*i.e.*, when it is cast to a CK_LONG, it holds -1).
2. Otherwise, if the specified attribute for the object is invalid (the object does not possess such an attribute), then the ulValueLen field in that triple is modified to hold the value -1.
3. Otherwise, if the pValue field has the value NULL, then the ulValueLen field is modified to hold the exact length of the specified attribute for the object.
4. Otherwise, if the length specified in ulValueLen is large enough to hold the value of the specified attribute for the object, then that attribute is copied into the buffer located at pValue, and the ulValueLen field is modified to hold the exact length of the attribute.
5. Otherwise, the ulValueLen field is modified to hold the value -1.

If case 1 applies to any of the requested attributes, then the call should return the value CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes, then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the requested attributes will CKR_OK be returned.

Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and CKR_BUFFER_TOO_SMALL do not denote true errors for C_GetAttributeValue. If a call to C_GetAttributeValue returns any of these three values, then the call must nonetheless have processed *every* attribute in the template supplied to C_GetAttributeValue. Each attribute in the template whose value *can be* returned by the call to C_GetAttributeValue *will be* returned by the call to C_GetAttributeValue.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hObject;
CK_BYTE*          pModulus, pExponent;

CK_ATTRIBUTE template[] =
{
  {CKA_MODULUS,         NULL, 0},
  {CKA_PUBLIC_EXPONENT, NULL, 0}
};
```

```
CK_RV rv;
...

rv = C_GetAttributeValue(hSession,
                         hObject,
                         &template,
                         2);
if (rv == CKR_OK) {
  pModulus = (CK_BYTE*)malloc(template[0].ulValueLen);
  template[0].pValue = pModulus;
  /* template[0].ulValueLen was set by C_GetAttributeValue */

  pExponent = (CK_BYTE*)malloc(template[1].ulValueLen);
  template[1].pValue = pExponent;
  /* template[1].ulValueLen was set by C_GetAttributeValue */

  rv = C_GetAttributeValue(hSession,
                           hObject,
                           &template,
                           2);
  if (rv == CKR_OK) {
    ...
  }
  free(pModulus);
  free(pExponent);
}
```

B4.5.5.5 C_FindObjectsInit

```
CK_RV C_FindObjectsInit(
           CK_SESSION_HANDLE hSession,
           CK_ATTRIBUTE*     pTemplate,
           CK_ULONG          ulCount
);
```

`C_FindObjectsInit` initializes a search for token objects that match a template. `hSession` is the session's handle; `pTemplate` points to a search template that specifies the attribute values to match; `ulCount` is the number of attributes in the search template. The matching criterion is an exact byte-for-byte match with all attributes in the template. To find all objects, set `ulCount` to 0.

After calling `C_FindObjectsInit`, the application may call `C_FindObjects` one or more times to obtain handles for objects matching the template, and then eventually call `C_FindObjectsFinal` to finish the active search operation. At most one search operation may be active at a given time in a given session.

The search support is limited in this Cryptographic API compared to standard Cryptoki:

- only token objects can be searched. Even if you set `ulCount` to 0, this will enumerate only token objects, not session objects
- you can only search on the `CKA_ID` attribute, the `CKA_CLASS` attribute, or both of them. Any other attribute template configuration will lead to a `CKR_ARGUMENTS_BAD`
- Objects are uniquely identified within the token by the combination of their `CKA_ID` attribute and `CKA_CLASS` attribute. As a consequence:
  - If the search template contains both the attribute `CKA_CLASS` and `CKA_ID`, the search will return at most one object
  - If the search template contains only the `CKA_CLASS` attribute, the search will return the list of the objects belonging to this class
  - If the search template contains only the `CKA_ID` attribute, the search will return at most four objects, each one belonging to a different object class
  - If the template contains no attribute or if `pTemplate` is `NULL`, then all token objects are returned.

If an operation is active within this session the function returns `CKR_OPERATION_ACTIVE`.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_FindObjectsFinal.

B4.5.5.6 C_FINDOBJECTS

```
CK_RV C_FindObjects(
           CK_SESSION_HANDLE    hSession,
           CK_OBJECT_HANDLE*    phObject,
           CK_ULONG             ulMaxObjectCount,
           CK_ULONG*            pulObjectCount
);
```

`C_FindObjects` continues a search for token objects that match a template, obtaining additional object handles. `hSession` is the session's handle; `phObject` points to the location that receives the list (array) of additional object handles; `ulMaxObjectCount` is the maximum number of object handles to be returned; `pulObjectCount` points to the location that receives the actual number of object handles returned.

If there are no more objects matching the template, then the location that `pulObjectCount` points to receives the value 0.

The search must have been initialized with `C_FindObjectsInit`.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_FindObjectsFinal.

B4.5.5.7 C_FINDOBJECTSFINAL

```
CK_RV C_FindObjectsFinal(
            CK_SESSION_HANDLE hSession
);
```

`C_FindObjectsFinal` terminates a search for token objects. `hSession` is the session's handle. Clean development expects any initialized Find to be terminated with a call to the corresponding Final.

Return values: `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_ULONG ulObjectCount;
CK_RV rv;
...

rv = C_FindObjectsInit(hSession, NULL, 0);
assert(rv == CKR_OK);

while (1) {
  rv = C_FindObjects(hSession,
                     &hObject,
                     1,
                     &ulObjectCount);

  if (rv != CKR_OK || ulObjectCount == 0) {
    break;
  }

  ...
}

rv = C_FindObjectsFinal(hSession);
assert(rv == CKR_OK);
```

B4.5.5.8 C_CLOSEOBJECTHANDLE

```
CK_RV C_CloseObjectHandle(
            CK_SESSION_HANDLE    hSession,
            CK_OBJECT_HANDLE     hObject
);
```

`C_CloseObjectHandle` closes an object handle. `hSession` is the session's handle and `hObject` is the object's handle.

An object handle must be closed as many times as it was opened for the underlying resources to be correctly released. More precisely, a handle reference count is incremented by one each time a handle on the object is returned to the caller by one of the functions `C_CreateObject`, `C_GenerateKey`, `C_GenerateKeyPair`, `C_DeriveKey`, `C_CopyObject`, or `C_FindObject`. The reference count is decremented by one each time `C_CloseObjectHandle` is called. If the reference count is zero then `C_CloseObjectHandle` deallocates the object resources.

The function `C_CloseObjectHandle` allows a fine control of the resources allocated for an object handle. In particular, if a search operation returns many object handles but only a few of them end up being used, it is advised to close the handles on all the other objects. Otherwise, some resources will stay allocated until the session is closed.

Internal API Note: the `SHandleClose` can also be used to close an object handle.

Return values:, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_ULONG ulObjectCount;
CK_RV rv;
...

rv = C_FindObjectsInit(hSession, NULL, 0);
assert(rv == CKR_OK);

while (1) {
  rv = C_FindObjects(hSession,
                     &hObject,
                     1,
                     &ulObjectCount);

  if (rv != CKR_OK || ulObjectCount == 0) {
    break;
  }

  ...

  rv = C_CloseObjectsHandle(hSession, hObject,
  assert(rv == CKR_OK);
}

rv = C_FindObjectsFinal(hSession);
assert(rv == CKR_OK);
```

B4.5.6 ENCRYPTION FUNCTIONS

Cryptoki provides the following functions for encrypting data:

B4.5.6.1 C_ENCRYPTINIT

```
CK_RV C_EncryptInit(
           CK_SESSION_HANDLE hSession,
           CK_MECHANISM*     pMechanism,
           CK_OBJECT_HANDLE  hKey
);
```

C_EncryptInit initializes an encryption operation. hSession is the session's handle; pMechanism points to the encryption mechanism; hKey is the handle of the encryption key.

The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, must be CK_TRUE.

After calling C_EncryptInit, the application can either call C_Encrypt to encrypt data in a single part; or call C_EncryptUpdate zero or more times, followed by C_EncryptFinal, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to C_Encrypt or C_EncryptFinal *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application must call C_EncryptInit again.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_SESSION_HANDLE_INVALID.

Example: see C_EncryptFinal.

B4.5.6.2 C_ENCRYPT

```
CK_RV C_Encrypt(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pData,
           CK_ULONG          ulDataLen,
           CK_BYTE*          pEncryptedData,
           CK_ULONG*         pulEncryptedDataLen
);
```

C_Encrypt encrypts single-part data. hSession is the session's handle; pData points to the data; ulDataLen is the length in bytes of the data; pEncryptedData points to the location that receives the encrypted data; pulEncryptedDataLen points to the location that holds the length in bytes of the encrypted data.

C_Encrypt uses the convention described in Section B4.5.2 on producing output.

The encryption operation must have been initialized with C_EncryptInit. A call to C_Encrypt always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

C_Encrypt can not be used to terminate a multi-part operation, and must be called after C_EncryptInit without intervening C_EncryptUpdate calls. If a call to C_EncryptUpdate has been performed after the call to C_EncryptInit, this function does nothing and returns CKR_OPERATION_NOT_INITIALIZED.

For some encryption mechanisms, the input plaintext data has certain length constraints (either because the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input data must consist of an integral number of blocks). If these constraints are not satisfied, then C_Encrypt will fail with return code CKR_DATA_LEN_RANGE.

The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if pData and pEncryptedData point to the same location.

For most mechanisms, C_Encrypt is equivalent to a sequence of C_EncryptUpdate operations followed by C_EncryptFinal.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_HANDLE_INVALID.

Example: see C_EncryptFinal for an example of similar functions.

B4.5.6.3 C_ENCRYPTUPDATE

```
CK_RV C_EncryptUpdate(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pPart,
           CK_ULONG          ulPartLen,
           CK_BYTE*          pEncryptedPart,
           CK_ULONG*         pulEncryptedPartLen
);
```

`C_EncryptUpdate` continues a multiple-part encryption operation, processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points to the location that holds the length in bytes of the encrypted data part.

`C_EncryptUpdate` uses the convention described in Section B4.5.2 on producing output.

The encryption operation must have been initialized with `C_EncryptInit`. This function may be called any number of times in succession. A call to `C_EncryptUpdate` which results in an error other than `CKR_BUFFER_TOO_SMALL` terminates the current encryption operation.

The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if `pPart` and `pEncryptedPart` point to the same location.

Any data pointed to by `pPart` may be modified by the caller once this function returns.

If the decryption operation that has been initialized is only supported in single-part, this function does nothing and returns `CKR_OPERATION_NOT_INITIALIZED`.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_EncryptFinal.

B4.5.6.4 C_ENCRYPTFINAL

```
CK_RV C_EncryptFinal(
            CK_SESSION_HANDLE hSession,
            CK_BYTE*          pLastEncryptedPart,
            CK_ULONG*         pulLastEncryptedPartLen
);
```

`C_EncryptFinal` finishes a multiple-part encryption operation. `hSession` is the session's handle; `pLastEncryptedPart` points to the location that receives the last encrypted data part, if any; `pulLastEncryptedPartLen` points to the location that holds the length of the last encrypted data part.

`C_EncryptFinal` uses the convention described in Section B4.5.2 on producing output.

The encryption operation must have been initialized with `C_EncryptInit`. A call to `C_EncryptFinal` always terminates the active encryption operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the ciphertext.

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism's input data must consist of an integral number of blocks. If these constraints are not satisfied, then `C_EncryptFinal` will fail with return code `CKR_DATA_LEN_RANGE`.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
#define PLAINTEXT_BUF_SZ  200
#define CIPHERTEXT_BUF_SZ 200

CK_ULONG          firstPieceLen, secondPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hKey;
CK_BYTE           iv[8];
CK_MECHANISM      mechanism =
{
  CKM_DES_CBC, iv, sizeof(iv)
};
CK_BYTE  data[PLAINTEXT_BUF_SZ];
CK_BYTE  encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulEncryptedData1Len;
CK_ULONG ulEncryptedData2Len;
CK_ULONG ulEncryptedData3Len;
CK_RV    rv;
...

firstPieceLen  = 90;
secondPieceLen = PLAINTEXT_BUF_SZ - firstPieceLen;
rv = C_EncryptInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {

  /* Encrypt first piece */
  ulEncryptedData1Len = sizeof(encryptedData);
  rv = C_EncryptUpdate(hSession,
                       &data[0],
                       firstPieceLen,
                       &encryptedData[0],
                       &ulEncryptedData1Len);
  if (rv != CKR_OK) {
    ...
  }

  /* Encrypt second piece */

  ulEncryptedData2Len = sizeof(encryptedData)-
      ulEncryptedData1Len;
  rv = C_EncryptUpdate(hSession,
                       &data[firstPieceLen],
                       secondPieceLen,
                       &encryptedData[ulEncryptedData1Len],
                       &ulEncryptedData2Len);
  if (rv != CKR_OK) {
    ...
  }

  /* Finalize the operation. There is no last encrypted part for the
     CKM_DES_CBC mechanism, but you must still pass a non-NULL buffer
     to properly finalize the operation */
  ulEncryptedData3Len = 0;
  rv = C_EncryptFinal(hSession,
                      &encryptedData[ulEncryptedData1Len +
                      ulEncryptedData2Len],
                      &ulEncryptedData3Len);

  if (rv != CKR_OK) {
    ...
  }
}
```

B4.5.7 Decryption functions

Cryptoki provides the following functions for decrypting data:

B4.5.7.1 C_DecryptInit

```
CK_RV C_DecryptInit(
           CK_SESSION_HANDLE hSession,
           CK_MECHANISM*     pMechanism,
           CK_OBJECT_HANDLE  hKey
);
```

C_DecryptInit initializes a decryption operation. hSession is the session's handle; pMechanism points to the decryption mechanism; hKey is the handle of the decryption key.

The CKA_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption, must be CK_TRUE.

After calling C_DecryptInit, the application can either call C_Decrypt to decrypt data in a single part; or call C_DecryptUpdate zero or more times, followed by C_DecryptFinal, to decrypt data in multiple parts. The decryption operation is active until the application uses a call to C_Decrypt or C_DecryptFinal *to actually obtain* the final piece of plaintext. To process additional data (in single or multiple parts), the application must call C_DecryptInit again

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_SESSION_HANDLE_INVALID.

Example: see C_DecryptFinal.

B4.5.7.2 C_DECRYPT

```
CK_RV C_Decrypt(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pEncryptedData,
           CK_ULONG          ulEncryptedDataLen,
           CK_BYTE*          pData,
           CK_ULONG*         pulDataLen
);
```

C_Decrypt decrypts encrypted data in a single part. hSession is the session's handle; pEncryptedData points to the encrypted data; ulEncryptedDataLen is the length of the encrypted data; pData points to the location that receives the recovered data; pulDataLen points to the location that holds the length of the recovered data.

C_Decrypt uses the convention described in Section B4.5.2 on producing output.

The decryption operation must have been initialized with C_DecryptInit. A call to C_Decrypt always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

C_Decrypt can not be used to terminate a multi-part operation, and must be called after C_DecryptInit without intervening C_DecryptUpdate calls. If a call to C_DecryptUpdate has been performed after the call to C_DecryptInit, this function does nothing and returns CKR_OPERATION_NOT_INITIALIZED.

The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if pEncryptedData and pData point to the same location.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

For most mechanisms, C_Decrypt is equivalent to a sequence of C_DecryptUpdate operations followed by C_DecryptFinal.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_HANDLE_INVALID.

Example: see C_DecryptFinal for an example of similar functions.

B4.5.7.3 C_DecryptUpdate

```
CK_RV C_DecryptUpdate(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pEncryptedPart,
           CK_ULONG          ulEncryptedPartLen,
           CK_BYTE*          pPart,
           CK_ULONG*         pulPartLen
);
```

`C_DecryptUpdate` continues a multiple-part decryption operation, processing another encrypted data part. `hSession` is the session's handle; `pEncryptedPart` points to the encrypted data part; `ulEncryptedPartLen` is the length of the encrypted data part; `pPart` points to the location that receives the recovered data part; `pulPartLen` points to the location that holds the length of the recovered data part.

`C_DecryptUpdate` uses the convention described in Section B4.5.2 on producing output.

The decryption operation must have been initialized with `C_DecryptInit`. This function may be called any number of times in succession. A call to `C_DecryptUpdate` which results in an error other than `CKR_BUFFER_TOO_SMALL` terminates the current decryption operation. If the decryption operation that has been initialized is only supported in single-part, this function does nothing and returns `CKR_OPERATION_NOT_INITIALIZED`.

The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if `pEncryptedPart` and `pPart` point to the same location.

`pEncryptedPart` may be modified by the caller once this function returns.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_ENCRYPTED_DATA_INVALID`, `CKR_ENCRYPTED_DATA_LEN_RANGE`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_DecryptFinal.

B4.5.7.4 C_DecryptFinal

```
CK_RV C_DecryptFinal(
            CK_SESSION_HANDLE hSession,
            CK_BYTE*          pLastPart,
            CK_ULONG*         pulLastPartLen
);
```

`C_DecryptFinal` finishes a multiple-part decryption operation. *hSession* is the session's handle; *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to the location that holds the length of the last recovered data part.

`C_DecryptFinal` uses the convention described in Section B4.5.2 on producing output.

The decryption operation must have been initialized with `C_DecryptInit`. A call to `C_DecryptFinal` always terminates the active decryption operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the plaintext.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either `CKR_ENCRYPTED_DATA_INVALID` or `CKR_ENCRYPTED_DATA_LEN_RANGE` may be returned.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_ENCRYPTED_DATA_INVALID`, `CKR_ENCRYPTED_DATA_LEN_RANGE`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
#define CIPHERTEXT_BUF_SZ 256
#define PLAINTEXT_BUF_SZ  256
CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hKey;
CK_BYTE           iv[8];
CK_MECHANISM      mechanism =
{
  CKM_DES_CBC, iv, sizeof(iv)
};
CK_BYTE  data[PLAINTEXT_BUF_SZ];
CK_BYTE  encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulData1Len, ulData2Len, ulData3Len;
CK_RV    rv;
...

firstEncryptedPieceLen  = 90;
secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ -
    firstEncryptedPieceLen;

rv = C_DecryptInit(hSession, &mechanism, hKey);

if (rv == CKR_OK) {
  /* Decrypt first piece */
  ulData1Len = sizeof(data);
  rv = C_DecryptUpdate(hSession,
                       &encryptedData[0],
                       firstEncryptedPieceLen,
                        &data[0],
                        &ulData1Len);
  if (rv != CKR_OK) {
    ...
  }

  /* Decrypt second piece */
  ulData2Len = sizeof(data)-ulData1Len;
  rv = C_DecryptUpdate(hSession,
                        &encryptedData[firstEncryptedPieceLen],
                        secondEncryptedPieceLen,
                        &data[ulData1Len],
                        &ulData2Len);

  if (rv != CKR_OK) {
    ...
  }

  /* Get last little decrypted bit */
  ulData3Len = sizeof(data) - ulData1Len - ulData2Len;
  rv = C_DecryptFinal(hSession,
                       &data[ulData1Len+ulData2Len],
                       &ulData3Len);

  if (rv != CKR_OK) {
    ...
  }
}
```

B4.5.8 MESSAGE DIGESTING FUNCTIONS

Cryptoki provides the following functions for digesting data:

B4.5.8.1 C_DIGESTINIT

```
CK_RV C_DigestInit(
           CK_SESSION_HANDLE hSession,
           CK_MECHANISM*     pMechanism
);
```

`C_DigestInit` initializes a message-digesting operation. `hSession` is the session's handle; `pMechanism` points to the digesting mechanism.

After calling `C_DigestInit`, the application can either call `C_Digest` to digest data in a single part; or call `C_DigestUpdate` zero or more times, followed by `C_DigestFinal`, to digest data in multiple parts. The message-digesting operation is active until the application uses a call to `C_Digest` or `C_DigestFinal` *to actually obtain* the message digest. To process additional data (in single or multiple parts), the application must call `C_DigestInit` again.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_MECHANISM_INVALID`, `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_DigestFinal.

B4.5.8.2 C_DIGEST

```
CK_RV C_Digest(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pData,
           CK_ULONG          ulDataLen,
           CK_BYTE*          pDigest,
           CK_ULONG*         pulDigestLen
);
```

`C_Digest` digests data in a single part. `hSession` is the session's handle, `pData` points to the data; `ulDataLen` is the length of the data; `pDigest` points to the location that receives the message digest; `pulDigestLen` points to the location that holds the length of the message digest.

`C_Digest` uses the convention described in Section B4.5.2 on producing output.

The digest operation must have been initialized with `C_DigestInit`. A call to `C_Digest` always terminates the active digest operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the message digest.

`C_Digest` can not be used to terminate a multi-part operation, and must be called after `C_DigestInit` without intervening `C_DigestUpdate` calls. If a call to `C_DigestUpdate` has been performed after the call to `C_DigestInit`, this function does nothing and returns `CKR_OPERATION_NOT_INITIALIZED`.

The input data and digest output can be in the same place, *i.e.*, it is OK if `pData` and `pDigest` point to the same location.

`C_Digest` is equivalent to a sequence of `C_DigestUpdate` operations followed by `C_DigestFinal`.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_DigestFinal for an example of similar functions.

B4.5.8.3 C_DigestUpdate

```
CK_RV C_DigestUpdate(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pPart,
           CK_ULONG          ulPartLen
);
```

`C_DigestUpdate` continues a multiple-part message-digesting operation, processing another data part. `hSession` is the session's handle, `pPart` points to the data part; `ulPartLen` is the length of the data part.

The message-digesting operation must have been initialized with `C_DigestInit`. A call to `C_DigestUpdate` which results in an error terminates the current digest operation.

Any data pointed to by `pPart` may be modified by the caller once this function returns.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_DigestFinal.

B4.5.8.4 C_DigestFinal

```
CK_RV C_DigestFinal(
            CK_SESSION_HANDLE hSession,
            CK_BYTE*          pDigest,
            CK_ULONG*         pulDigestLen
);
```

`C_DigestFinal` finishes a multiple-part message-digesting operation, returning the message digest. `hSession` is the session's handle; `pDigest` points to the location that receives the message digest; `pulDigestLen` points to the location that holds the length of the message digest.

C_DigestFinal uses the convention described in Section B4.5.2 on producing output.

The digest operation must have been initialized with `C_DigestInit`. A call to `C_DigestFinal` always terminates the active digest operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the message digest.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
CK_SESSION_HANDLE hSession;
CK_MECHANISM mechanism =
{
  CKM_MD5, NULL*   , 0
};
CK_BYTE data[] = {...};
CK_BYTE digest[16];
CK_ULONG ulDigestLen;
CK_RV rv;
...

rv = C_DigestInit(hSession, &mechanism);
if (rv != CKR_OK) {
  ...
}

rv = C_DigestUpdate(hSession, data, sizeof(data));
if (rv != CKR_OK) {
  ...
}

ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
...
```

B4.5.9 SIGNING AND MACING FUNCTIONS

Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations also encompass message authentication codes):

B4.5.9.1 C_SIGNINIT

```
CK_RV C_SignInit(
           CK_SESSION_HANDLE hSession,
           CK_MECHANISM*      pMechanism,
           CK_OBJECT_HANDLE   hKey
);
```

`C_SignInit` initializes a signature operation, where the signature is an appendix to the data. `hSession` is the session's handle; `pMechanism` points to the signature mechanism; `hKey` is the handle of the signature key.

The `CKA_SIGN` attribute of the signature key, which indicates whether the key supports signatures, must be `CK_TRUE`.

After calling `C_SignInit`, the application can either call `C_Sign` to sign in a single part; or call `C_SignUpdate` one or more times, followed by `C_SignFinal`, to sign data in multiple parts. The signature operation is active until the application uses a call to `C_Sign` or `C_SignFinal` *to actually obtain* the signature. To process additional data (in single or multiple parts), the application must call `C_SignInit` again.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_KEY_FUNCTION_NOT_PERMITTED`, `CKR_KEY_HANDLE_INVALID`, `CKR_KEY_SIZE_RANGE`, `CKR_KEY_TYPE_INCONSISTENT`, `CKR_MECHANISM_INVALID`, `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_SignFinal.

B4.5.9.2 C_SIGN

```
CK_RV C_Sign(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pData,
           CK_ULONG          ulDataLen,
           CK_BYTE*          pSignature,
           CK_ULONG*         pulSignatureLen
);
```

`C_Sign` signs data in a single part, where the signature is an appendix to the data. `hSession` is the session's handle; `pData` points to the data; `ulDataLen` is the length of the data; `pSignature` points to the location that receives the signature; `pulSignatureLen` points to the location that holds the length of the signature.

`C_Sign` uses the convention described in Section B4.5.2 on producing output.

The signing operation must have been initialized with `C_SignInit`. A call to `C_Sign` always terminates the active signing operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the signature.

`C_Sign` can not be used to terminate a multi-part operation, and must be called after `C_SignInit` without intervening `C_SignUpdate` calls. If a call to `C_SignUpdate` has been performed after the call to `C_SignInit`, this function does nothing and returns `CKR_OPERATION_NOT_INITIALIZED`.

For most mechanisms, `C_Sign` is equivalent to a sequence of `C_SignUpdate` operations followed by `C_SignFinal`.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_INVALID`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_SignFinal for an example of similar functions.

B4.5.9.3 C_SIGNUPDATE

```
CK_RV C_SignUpdate(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pPart,
           CK_ULONG          ulPartLen
);
```

C_SignUpdate continues a multiple-part signature operation, processing another data part. hSession is the session's handle, pPart points to the data part; ulPartLen is the length of the data part.

The signature operation must have been initialized with C_SignInit. This function may be called any number of times in succession. A call to C_SignUpdate which results in an error terminates the current signature operation. If the signature operation that has been initialized is only supported in single-part, this function does nothing and returns CKR_OPERATION_NOT_INITIALIZED.

Any data pointed to by pPart may be modified by the caller once this function returns.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_HANDLE_INVALID.

Example: see C_SignFinal.

B4.5.9.4 C_SignFinal

```
CK_RV C_SignFinal(
            CK_SESSION_HANDLE hSession,
            CK_BYTE*          pSignature,
            CK_ULONG*         pulSignatureLen
);
```

`C_SignFinal` finishes a multiple-part signature operation, returning the signature. `hSession` is the session's handle; `pSignature` points to the location that receives the signature; `pulSignatureLen` points to the location that holds the length of the signature.

`C_SignFinal` uses the convention described in Section B4.5.2 on producing output.

The signing operation must have been initialized with `C_SignInit`. A call to `C_SignFinal` always terminates the active signing operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the signature.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
CK_SESSION_HANDLE      hSession;
CK_OBJECT_HANDLE       hKey;
CK_RSA_PKCS_PSS_PARAMS params =
{
  CKM_SHA_1, CKG_MGF1_SHA1, 20
};
CK_MECHANISM mechanism = {
  CKM_SHA1_RSA_PKCS_PSS, &params, sizeof(params)
};
CK_BYTE  data[] = {...};
CK_BYTE  signature[128];
CK_ULONG ulSignature;
CK_RV    rv;
...

rv = C_SignInit(hSession, &mechanism, hKey);

if (rv == CKR_OK) {
  rv = C_SignUpdate(hSession, data, sizeof(data));
  ...
  ulSignature = sizeof(signature);
  rv = C_SignFinal(hSession, signature, &ulSignature);
  ...
}
```

B4.5.10 Functions for verifying signatures and MACs

Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki, these operations also encompass message authentication codes):

B4.5.10.1 C_VerifyInit

```
CK_RV C_VerifyInit(
           CK_SESSION_HANDLE hSession,
           CK_MECHANISM*     pMechanism,
           CK_OBJECT_HANDLE  hKey
);
```

`C_VerifyInit` initializes a verification operation, where the signature is an appendix to the data. `hSession` is the session's handle; `pMechanism` points to the structure that specifies the verification mechanism; `hKey` is the handle of the verification key.

The `CKA_VERIFY` attribute of the verification key, which indicates whether the key supports verification where the signature is an appendix to the data, must be `CK_TRUE`.

After calling `C_VerifyInit`, the application can either call `C_Verify` to verify a signature on data in a single part; or call `C_VerifyUpdate` one or more times, followed by `C_VerifyFinal`, to verify a signature on data in multiple parts. The verification operation is active until the application calls `C_Verify` or `C_VerifyFinal`. To process additional data (in single or multiple parts), the application must call `C_VerifyInit` again.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_KEY_FUNCTION_NOT_PERMITTED`, `CKR_KEY_HANDLE_INVALID`, `CKR_KEY_SIZE_RANGE`, `CKR_KEY_TYPE_INCONSISTENT`, `CKR_MECHANISM_INVALID`, `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_VerifyFinal.

B4.5.10.2 C_VERIFY

```
CK_RV C_Verify(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pData,
           CK_ULONG          ulDataLen,
           CK_BYTE*          pSignature,
           CK_ULONG          ulSignatureLen
);
```

`C_Verify` verifies a signature in a single-part operation, where the signature is an appendix to the data. `hSession` is the session's handle; `pData` points to the data; `ulDataLen` is the length of the data; `pSignature` points to the signature; `ulSignatureLen` is the length of the signature.

The verification operation must have been initialized with `C_VerifyInit`. A call to `C_Verify` always terminates the active verification operation.

A successful call to `C_Verify` should return either the value `CKR_OK` (indicating that the supplied signature is valid) or `CKR_SIGNATURE_INVALID` (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then `CKR_SIGNATURE_LEN_RANGE` should be returned. In any of these cases, the active signing operation is terminated.

`C_Verify` can not be used to terminate a multi-part operation, and must be called after `C_VerifyInit` without intervening `C_VerifyUpdate` calls. If a call to `C_VerifyUpdate` has been performed after the call to `C_VerifyInit`, this function does nothing and returns `CKR_OPERATION_NOT_INITIALIZED`.

For most mechanisms, `C_Verify` is equivalent to a sequence of `C_VerifyUpdate` operations followed by `C_VerifyFinal`.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_INVALID`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SIGNATURE_INVALID`, `CKR_SIGNATURE_LEN_RANGE`.

Example: see C_VerifyFinal for an example of similar functions.

B4.5.10.3 C_VERIFYUPDATE

```
CK_RV C_VerifyUpdate(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pPart,
           CK_ULONG          ulPartLen
);
```

`C_VerifyUpdate` continues a multiple-part verification operation, processing another data part. `hSession` is the session's handle, `pPart` points to the data part; `ulPartLen` is the length of the data part.

The verification operation must have been initialized with `C_VerifyInit`. This function may be called any number of times in succession. A call to `C_VerifyUpdate` which results in an error terminates the current verification operation. If the verification operation that has been initialized is only supported in single-part, this function does nothing and returns `CKR_OPERATION_NOT_INITIALIZED`.

Any data pointed to by `pPart` may be modified by the caller once this function returns.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_VerifyFinal.

B4.5.10.4 C_VERIFYFINAL

```
CK_RV C_VerifyFinal(
            CK_SESSION_HANDLE hSession,
            CK_BYTE*          pSignature,
            CK_ULONG          ulSignatureLen
);
```

`C_VerifyFinal` finishes a multiple-part verification operation, checking the signature. `hSession` is the session's handle; `pSignature` points to the signature; `ulSignatureLen` is the length of the signature.

The verification operation must have been initialized with `C_VerifyInit`. A call to `C_VerifyFinal` always terminates the active verification operation.

A successful call to `C_VerifyFinal` should return either the value `CKR_OK` (indicating that the supplied signature is valid) or `CKR_SIGNATURE_INVALID` (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then `CKR_SIGNATURE_LEN_RANGE` should be returned. In any of these cases, the active verifying operation is terminated.

Internal API Note

If a Cryptoki Update Shortcut is active on the operation and if this function terminates the operation, then the shortcut is automatically deactivated. See section B2.3.14.1, "*Cryptoki Update Shortcut*" for more details.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SIGNATURE_INVALID`, `CKR_SIGNATURE_LEN_RANGE`.

Example:

```
CK_SESSION_HANDLE      hSession;
CK_OBJECT_HANDLE       hKey;
CK_RSA_PKCS_PSS_PARAMS params =
{
  CKM_SHA_1, CKG_MGF1_SHA1, 20
};
CK_MECHANISM mechanism =
{
  CKM_SHA1_RSA_PKCS_PSS, &params, sizeof(params)
};
CK_BYTE data[] = {...};
CK_BYTE signature[128];
CK_RV   rv;
...

rv = C_VerifyInit(hSession, &mechanism, hKey);

if (rv == CKR_OK) {
  rv = C_VerifyUpdate(hSession, data, sizeof(data));
  ...
  rv = C_VerifyFinal(hSession,signature, sizeof(signature));
  ...
}
```

B4.5.11 Key management functions

Cryptoki provides the following functions for key management:

B4.5.11.1 C_GenerateKey

```
CK_RV C_GenerateKey(
              CK_SESSION_HANDLE     hSession
              CK_MECHANISM*         pMechanism,
              CK_ATTRIBUTE*         pTemplate,
              CK_ULONG              ulCount,
              CK_OBJECT_HANDLE*     phKey
);
```

C_GenerateKey generates a secret key creating a new object. hSession is the session's handle; pMechanism points to the generation mechanism; pTemplate points to the template for the new key; ulCount is the number of attributes in the template; phKey points to the location that receives the handle of the new key or set of domain parameters.

Since the type of key is implicit in the generation mechanism, the template does not need to supply a key type. If it does supply a key type which is inconsistent with the generation mechanism, C_GenerateKey fails and returns the error code CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly: it must be omitted or set to CKO_SECRET_KEY.

If a call to C_GenerateKey cannot support the precise template supplied to it, it will fail and return without creating an object.

Only session objects can be generated during a read-only session. If an attempt is made to generate a token object in a read-only session, this function returns CKR_SESSION_READ_ONLY.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hKey;
CK_MECHANISM      mechanism =
{
  CKM_DES_KEY_GEN, NULL, 0
};
CK_BYTE           objId[2] = {0x00, 0x01};
CK_BBOOL          bTrue = TRUE;

CK_ATTRIBUTE      keyTemplate[3] =
{
  {CKA_ID,        (CK_BYTE*)objId,      2               },
  {CKA_TOKEN,     &bTrue,               sizeof(bTrue)   },
  {CKA_ENCRYPT,   &bTrue,               sizeof(bTrue)   }
};
CK_RV             rv;
...

rv = C_GenerateKey(hSession,
                    &mechanism,
                    keyTemplate,
                    2,
                    &hKey);

if (rv == CKR_OK) {
  ...
}
```

B4.5.11.2 C_GENERATEKEYPAIR

```
CK_RV C_GenerateKeyPair(
            CK_SESSION_HANDLE    hSession,
            CK_MECHANISM*        pMechanism,
            CK_ATTRIBUTE*        pPublicKeyTemplate,
            CK_ULONG             ulPublicKeyAttributeCount,
            CK_ATTRIBUTE*        pPrivateKeyTemplate,
            CK_ULONG             ulPrivateKeyAttributeCount,
            CK_OBJECT_HANDLE*    phPublicKey,
            CK_OBJECT_HANDLE*    phPrivateKey
);
```

`C_GenerateKeyPair` generates a public/private key pair, creating new key objects. `hSession` is the session's handle; `pMechanism` points to the key generation mechanism; `pPublicKeyTemplate` points to the template for the public key; `ulPublicKeyAttributeCount` is the number of attributes in the public-key template; `pPrivateKeyTemplate` points to the template for the private key; `ulPrivateKeyAttributeCount` is the number of attributes in the private-key template; `phPublicKey` points to the location that receives the handle of the new public key; `phPrivateKey` points to the location that receives the handle of the new private key.

Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates do not need to supply key types. If one of the templates does supply a key type which is inconsistent with the key generation mechanism, `C_GenerateKeyPair` fails and returns the error code `CKR_TEMPLATE_INCONSISTENT`. The `CKA_CLASS` attribute is treated similarly.

If a call to `C_GenerateKeyPair` cannot support the precise templates supplied to it, it will fail and return without creating any key objects.

A call to `C_GenerateKeyPair` will never create just one key and return. A call can fail, and create no keys; or it can succeed, and create a matching public/private key pair.

Only session objects can be generated during a read-only session. If an attempt is made to generate token objects in a read-only session, this function returns `CKR_SESSION_READ_ONLY`.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_ATTRIBUTE_TYPE_INVALID`, `CKR_ATTRIBUTE_VALUE_INVALID`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DOMAIN_PARAMS_INVALID`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_MECHANISM_INVALID`, `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SESSION_READ_ONLY`, `CKR_TEMPLATE_INCOMPLETE`, `CKR_TEMPLATE_INCONSISTENT`.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hPublicKey, hPrivateKey;
CK_MECHANISM mechanism =
{
  CKM_RSA_PKCS_KEY_PAIR_GEN, NULL, 0
};

CK_ULONG modulusBits       = 768;
CK_BYTE  publicExponent[] = { 3 };
CK_BYTE  subject[]         = {...};

CK_ATTRIBUTE publicKeyTemplate[] =
{
  {CKA_ENCRYPT,         &true,          sizeof(true)},
  {CKA_VERIFY,          &true,          sizeof(true)},
  {CKA_MODULUS_BITS,    &modulusBits,   sizeof(modulusBits)},
  {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
};
```

```
CK_ATTRIBUTE privateKeyTemplate[] =
{
  {CKA_DECRYPT, &true,  sizeof(true)},
  {CKA_SIGN,    &true,  sizeof(true)},
};
CK_RV rv;

rv = C_GenerateKeyPair(hSession,
                       &mechanism,
                       publicKeyTemplate,
                       4,
                       privateKeyTemplate,
                       2,
                       &hPublicKey,
                       &hPrivateKey);
if (rv == CKR_OK) {
  ...
}
```

B4.5.11.3 C_DERIVEKEY

```
CK_RV C_DeriveKey(
            CK_SESSION_HANDLE    hSession,
            CK_MECHANISM*        pMechanism,
            CK_OBJECT_HANDLE     hBaseKey,
            CK_ATTRIBUTE*        pTemplate,
            CK_ULONG             ulAttributeCount,
            CK_OBJECT_HANDLE*    phKey
);
```

C_DeriveKey derives a key from a base key, creating a new key object. hSession is the session's handle; pMechanism points to a structure that specifies the key derivation mechanism; hBaseKey is the handle of the base key; pTemplate points to the template for the new key; ulAttributeCount is the number of attributes in the template; and phKey points to the location that receives the handle of the derived key.

If a call to C_DeriveKey cannot support the precise template supplied to it, it will fail and return without creating any key object.

Only session objects can be derived during a read-only session. If an attempt is made to derive token objects in a read-only session, this function returns CKR_SESSION_READ_ONLY.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE  hPublicKey, hPrivateKey, hKey;
CK_MECHANISM      keyPairMechanism =
{
  CKM_DH_PKCS_KEY_PAIR_GEN, NULL, 0
};
CK_BYTE      prime[]     = {...};
CK_BYTE      base[]      = {...};
CK_BYTE      publicValue[128];
CK_BYTE      otherPublicValue[128];
CK_MECHANISM mechanism =
{
  CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)
};
CK_ATTRIBUTE pTemplate[] =
{
  CKA_VALUE, &publicValue, sizeof(publicValue)}
};
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE     keyType  = CKK_GENERIC_SECRET;
CK_BBOOL        true     = CK_TRUE;
CK_BBOOL        false    = CK_FALSE;
CK_ATTRIBUTE publicKeyTemplate[] =
{
  {CKA_PRIME, prime, sizeof(prime)},
  {CKA_BASE,  base,  sizeof(base)}
};
CK_ATTRIBUTE privateKeyTemplate[] =
{
  {CKA_DERIVE, &true, sizeof(true)},
  {CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY, &true, sizeof(true)}
```

```
};
CK_ATTRIBUTE template[] =
{
  {CKA_CLASS,     &keyClass, sizeof(keyClass)},
  {CKA_KEY_TYPE,  &keyType,  sizeof(keyType)},
  {CKA_SENSITIVE, &false,    sizeof(false)},
  {CKA_DECRYPT,   &true,     sizeof(true)}
};
CK_RV rv;

rv = C_GenerateKeyPair(hSession,
                       &keyPairMechanism,
                       publicKeyTemplate,
                       2,
                       privateKeyTemplate,
                       1,
                       &hPublicKey,
                       &hPrivateKey);
if (rv == CKR_OK) {
  rv = C_GetAttributeValue(hSession, hPublicKey,
                           &pTemplate, 1);
  /* send your public key to the other party */
  ...
  if (rv == CKR_OK) {
    /* Put other guy's public value in otherPublicValue */
    ...
    rv = C_DeriveKey( hSession, &mechanism,
                      hPrivateKey, template,
                      4, &hKey);
    if (rv == CKR_OK) {
      ...
    }
  }
}
```

B4.5.12 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

B4.5.12.1 C_SeedRandom

```
CK_RV C_SeedRandom(
           CK_SESSION_HANDLE hSession,
           CK_BYTE*          pSeed,
           CK_ULONG          ulSeedLen
);
```

`C_SeedRandom` mixes additional seed material into the token's random number generator. `hSession` is the session's handle; `pSeed` points to the seed material; and `ulSeedLen` is the length in bytes of the seed material.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_RANDOM_SEED_NOT_SUPPORTED`, `CKR_RANDOM_NO_RNG`, `CKR_SESSION_HANDLE_INVALID`.

Example: see C_GenerateRandom.

B4.5.12.2 C_GENERATERANDOM

```
CK_RV C_GenerateRandom(
            CK_SESSION_HANDLE hSession,
            CK_BYTE*          pRandomData,
            CK_ULONG          ulRandomLen
);
```

C_GenerateRandom generates random or pseudo-random data. hSession is the session's handle; pRandomData points to the location that receives the random data; and ulRandomLen is the length in bytes of the random or pseudo-random data to be generated.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_RANDOM_NO_RNG, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_BYTE seed[] = {...};
CK_BYTE randomData[] = {...};
CK_RV rv;
...

rv = C_SeedRandom(hSession, seed, sizeof(seed));
if (rv != CKR_OK) {
  ...
}
rv = C_GenerateRandom(hSession, randomData,sizeof(randomData));
if (rv == CKR_OK) {
  ...
}
```

B4.6 CRYPTOKI MECHANISMS

This section describes all the Cryptoki mechanisms supported in the Cryptographic API.

B4.6.1 RSA

This section defines the RSA key type "`CKK_RSA`" for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of RSA key objects.

This section defines the following mechanisms:

```
CKM_RSA_PKCS_KEY_PAIR_GEN
CKM_RSA_PKCS
CKM_RSA_X_509
CKM_MD5_RSA_PKCS
CKM_SHA1_RSA_PKCS
CKM_SHA224_RSA_PKCS
CKM_SHA256_RSA_PKCS
CKM_SHA384_RSA_PKCS
CKM_SHA512_RSA_PKCS
CKM_RSA_PKCS_OAEP
CKM_RSA_PKCS_PSS
CKM_SHA1_RSA_PKCS_PSS
CKM_SHA224_RSA_PKCS_PSS
CKM_SHA256_RSA_PKCS_PSS
CKM_SHA384_RSA_PKCS_PSS
CKM_SHA512_RSA_PKCS_PSS
```

B4.6.1.1 RSA PUBLIC KEY OBJECTS

RSA public key objects (object class **CKO_PUBLIC_KEY,** key type **CKK_RSA**) hold RSA public keys. The following table defines the RSA public key object attributes:

Table B4-14, RSA Public Key Object Attributes

Attribute	Data type	Meaning
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_PUBLIC_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_RSA`
`CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_SENSITIVE, CKA_ENCRYPT, CKA_VERIFY, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_MODULUS`	Big integer	Modulus *n*
`CKA_MODULUS_BITS`	`CK_ULONG`	Length in bits of modulus *n*
`CKA_PUBLIC_EXPONENT`	Big integer	Public exponent *e*

All these attributes can always be retrieved with the function `C_GetAttributeValue` as public key objects are never sensitive.

Note that when creating a public RSA key, the attribute `CKA_MODULUS_BITS` must not be specified. The implementation computes the size in bits of the modulus as part of the object creation.

When generating a RSA key pair, the attribute `CKA_MODULUS_BITS` must be specified in the template of the public key, but not in the template of the private key.

B4.6.1.2 RSA PRIVATE KEY OBJECTS

RSA private key objects (object class `CKO_PRIVATE_KEY`, key type `CKK_RSA`) hold RSA private keys. The following table defines the RSA private key object attributes, in addition to the common attributes defined for this object class:

Table B4-15, RSA Private Key Object Attributes

Attribute	**Data type**	**Meaning**
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_PRIVATE_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_RSA`
`CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_SENSITIVE, CKA_DECRYPT, CKA_SIGN, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*" Note that if your product implements the Developer APIs V3.1 or later, the attribute `CKA_SENSITIVE` may be set to `CK_FALSE` when the object is created or generated.	
`CKA_MODULUS`	Big integer	Modulus *n*
`CKA_PUBLIC_EXPONENT`	Big integer	Public exponent *e*
`CKA_PRIVATE_EXPONENT`	Big integer	Private exponent *d*
`CKA_PRIME_1`	Big integer	Prime *p*
`CKA_PRIME_2`	Big integer	Prime *q*
`CKA_EXPONENT_1`	Big integer	Private exponent *d* modulo *p-1*
`CKA_EXPONENT_2`	Big integer	Private exponent *d* module *q-1*
`CKA_COEFFICIENT`	Big integer	CRT coefficient q^{-1} mod *p*

When creating a RSA private key object, the attributes `CKA_MODULUS`, `CKA_PUBLIC_EXPONENT`, and `CKA_PRIVATE_EXPONENT` must always be provided. The attributes `CKA_PRIME_1`, `CKA_PRIME_2`, `CKA_EXPONENT_1`, `CKA_EXPONENT_2`, and `CKA_COEFFICIENT` are optional in the sense that the caller must provide either all of them or none. In some implementations, providing these additional attributes can accelerate the operations using the private key.

The attributes `CKA_PRIVATE_EXPONENT`, `CKA_PRIME_1`, `CKA_PRIME_2`, `CKA_EXPONENT_1`, `CKA_EXPONENT_2`, and `CKA_COEFFICIENT` are protected. By default, a RSA private key is sensitive and these attributes cannot be retrieved using the function `C_GetAttributeValue`. However, if your product implements the Developer APIs V3.1 or later, a RSA private key object can be taggued as non-sensitive by setting the attribute `CKA_SENSITIVE` to `CK_FALSE` in the template passed to the function `C_CreateObject` or `C_GenerateKeyPair`. In this case, the protected attributes can later be retrieved using the function `C_GetAttributeValue`. If the attribute `CKA_SENSITIVE` is not specified or set to `CK_TRUE`, then the protected attributes cannot be retrieved.

Creating or generating a non-sensitive RSA private key is not possible in the Developer APIs V3.0. A call to `C_CreateObject` or `C_GenerateKeyPair` that sets the attribute `CKA_SENSITIVE` even to `CK_TRUE` fails with the error code `CKR_ATTRIBUTE_TYPE_INVALID`.

Note that when generating an RSA private key, there is no `CKA_MODULUS_BITS` attribute specified. This is because RSA private keys are only generated as part of an RSA key *pair*, and the `CKA_MODULUS_BITS` attribute for the pair is specified in the template for the RSA public key.

B4.6.1.3 PKCS #1 RSA KEY PAIR GENERATION

The PKCS #1 RSA key pair generation mechanism, denoted `CKM_RSA_PKCS_KEY_PAIR_GEN`, is a key pair generation mechanism based on the RSA public-key cryptosystem, as defined in **[PKCS#1]**.

It does not have any parameters.

The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public exponent, as specified in the `CKA_MODULUS_BITS` and `CKA_PUBLIC_EXPONENT` attributes of the template for the public key. The `CKA_PUBLIC_EXPONENT` may be omitted in which case the mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537).

When the caller requests the generation of a RSA key pair with a modulus size of *2N* bits, the implementation is allowed to generate a modulus with *2N-1* actual bits. This is because the implementation may simply generate two prime numbers of *N* bits and the modulus, being the product of these two primes, may have only *2N-1* bits. In this case, the implementation is not required to retry the key pair generation until a *2N*-bit modulus is found.

Similarly, when requested to generate a key-pair with *2N-1* bits, the implementation can return a *2N*-bit modulus.

In these cases, the implementation will update the value of the `CKA_MODULUS_BITS` attribute and the caller can subsequently get the actual size of the modulus by using the function `C_GetAttributeValue`. The implementation does not update the template passed to the function `C_GenerateKeyPair` because this template is read-only.

B4.6.1.4 PKCS #1 v1.5 RSA

The PKCS #1 v1.5 RSA mechanism, denoted `CKM_RSA_PKCS`, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports single-part encryption and decryption. This mechanism corresponds only to the part of PKCS #1 v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for the `md5withRSAEncryption` algorithms in PKCS #1 v1.5.

This mechanism does not have a parameter.

Constraints on key types and the length of the data are summarized in the following table. For encryption, decryption, the input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

Table B4-16: PKCS #1 v1.5 RSA: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`[1]	RSA public key	≤ *k*-11	*k*	block type `02`
`C_Decrypt`[1]	RSA private key	*k*	≤ *k*-11	block type `02`

[1] Single-part operations only.

B4.6.1.5 PKCS #1 RSA OAEP MECHANISM PARAMETERS

B4.6.1.5.1 CK_RSA_PKCS_MGF_TYPE

`CK_RSA_PKCS_MGF_TYPE` is used to indicate the Message Generation Function (MGF) applied to a message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the PKCS #1 PSS signature scheme. It is defined as follows:

```
typedef CK_ULONG CK_RSA_PKCS_MGF_TYPE;
```

The following MGFs are defined in PKCS #1. The following table lists the defined functions.

Table B4-17, PKCS #1 Mask Generation Functions

Source Identifier	Value
CKG_MGF1_SHA1	0x00000001
CKG_MGF1_SHA224	0x00000005
CKG_MGF1_SHA256	0x00000002
CKG_MGF1_SHA384	0x00000003
CKG_MGF1_SHA512	0x00000004

B4.6.1.5.2 CK_RSA_PKCS_OAEP_SOURCE_TYPE

CK_RSA_PKCS_OAEP_SOURCE_TYPE is used to indicate the source of the encoding parameter when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

```
typedef CK_ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;
```

The following encoding parameter sources are defined in PKCS #1. The following table lists the defined sources along with the corresponding data type for the pSourceData field in the CK_RSA_PKCS_OAEP_PARAMS structure defined below.

Table B4-18: PKCS #1 RSA OAEP : Encoding Parameter sources

Source Identifier	Value	Data Type
CKZ_DATA_SPECIFIED	0x00000001	Array of CK_BYTE containing the value of the encoding parameter. If the parameter is empty, pSourceData must be NULL and ulSourceDataLen must be zero.

B4.6.1.5.3 CK_RSA_PKCS_OAEP_PARAMS

CK_RSA_PKCS_OAEP_PARAMS is a structure that provides the parameters to the CKM_RSA_PKCS_OAEP mechanism. The structure is defined as follows:

```
typedef struct CK_RSA_PKCS_OAEP_PARAMS {
 CK_MECHANISM_TYPE            hashAlg;
 CK_RSA_PKCS_MGF_TYPE         mgf;
 CK_RSA_PKCS_OAEP_SOURCE_TYPE source;
 void*                       pSourceData;
 CK_ULONG                     ulSourceDataLen;
} CK_RSA_PKCS_OAEP_PARAMS;
```

The fields of the structure have the following meanings:

- hashAlg: mechanism ID of the Message digest algorithm used to calculate the digest of the encoding parameter
- mgf: mask generation function to use on the encoded block
- source: source of the encoding parameter
- pSourceData: data used as the input for the encoding parameter source
- ulSourceDataLen: length of the encoding parameter source input

The value of hashAlg can only be one of the following:

- CKM_SHA_1
- CKM_SHA224

- `CKM_SHA256`
- `CKM_SHA384`
- `CKM_SHA512`

The value of the parameter mgf can be any of the values defined in section B4.6.1.5.1, "*CK_RSA_PKCS_MGF_TYPE*".

B4.6.1.6 PKCS #1 RSA OAEP

The PKCS #1 RSA OAEP mechanism, denoted `CKM_RSA_PKCS_OAEP`, is a multipurpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1. It supports single-part encryption and decryption.

It has a parameter, a `CK_RSA_PKCS_OAEP_PARAMS` structure, defined in section B4.6.1.5.3.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, `k` is the length in bytes of the RSA modulus, and `hLen` is the output length of the message digest algorithm specified by the `hashAlg` field of the `CK_RSA_PKCS_OAEP_PARAMS` structure.

Table B4-19: PKCS #1 RSA OAEP: Key and Data Length

Function	Key type	Input length	Output length
`C_Encrypt`[1]	RSA public key	$\leq k$-2-2*hLen*	*k*
`C_Decrypt`[1]	RSA private key	$\leq k$	$\leq k$-2-2*hLen*

[1] Single-part operations only.

B4.6.1.7 PKCS #1 RSA PSS MECHANISM PARAMETERS

B4.6.1.7.1 CK_RSA_PKCS_PSS_PARAMS

`CK_RSA_PKCS_PSS_PARAMS` is a structure that provides the parameters to the `CKM_RSA_PKCS_PSS` mechanism. The structure is defined as follows:

```
typedef struct CK_RSA_PKCS_PSS_PARAMS {
     CK_MECHANISM_TYPE    hashAlg;
     CK_RSA_PKCS_MGF_TYPE mgf;
     CK_ULONG             sLen;
} CK_RSA_PKCS_PSS_PARAMS;
```

The fields of the structure have the following meanings:

- `hashAlg`: hash algorithm used in the PSS encoding; if the signature mechanism does not include message hashing, then this value must be the mechanism used by the application to generate the message hash; if the signature mechanism includes hashing, then this value must match the hash algorithm indicated by the signature mechanism
- `mgf`: mask generation function to use on the encoded block
- `sLen`: length, in bytes, of the salt value used in the PSS encoding; typical values are the length of the message hash and zero

The value of `hashAlg` can only be one of the following:

- `CKM_SHA_1`
- `CKM_SHA224`
- `CKM_SHA256`
- `CKM_SHA384`
- `CKM_SHA512`

The value of the parameter mgf can be any of the values defined in section B4.6.1.5.1, "*CK_RSA_PKCS_MGF_TYPE*".

B4.6.1.8 PKCS #1 RSA PSS

The PKCS #1 RSA PSS mechanism, denoted `CKM_RSA_PKCS_PSS`, is a mechanism based on the RSA public-key cryptosystem and the PSS block format defined in **[PKCS#1]** It supports single-part signature generation and verification without message recovery. This mechanism corresponds only to the part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash value on the message to be signed.

It has a parameter, a `CK_RSA_PKCS_PSS_PARAMS` structure, which is defined in section B4.6.1.7.1. The *sLen* field must be less than or equal to *k**-2-*hLen* and *hLen* is the length of the input to the C_Sign or C_Verify function. *k** is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case *k** is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table B4-20: PKCS #1 RSA PSS: Key and Data Length

Function	Key type	Input length	Output length
`C_Sign`^{1}	RSA private key	*hLen*	*k*
`C_Verify`^{1}	RSA public key	*hLen*, *k* (digest following by the signature)	N/A

$^{(1)}$ Single-part operations only.

B4.6.1.9 X.509 (RAW) RSA

The X.509 (raw) RSA mechanism, denoted `CKM_RSA_X_509`, is a multi-purpose mechanism based on the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures and verification without message recovery. All these operations are based on so-called “raw” RSA, as assumed in X.509.

“Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first, applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first. The input string, considered as an integer, must be less than the modulus; the output string is also less than the modulus.

This mechanism does not have a parameter.

Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism, padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the sequence of plaintext bytes b_1 b_2 … b_n ($n \le k$), Cryptoki forms $P=2^{n-1}b_1+2^{n-2}b_2+\ldots+b_n$. This number must be less than the RSA modulus. The *k*-byte ciphertext (*k* is the length in bytes of the RSA modulus) is produced by raising P to the RSA public exponent modulo the RSA modulus. Decryption of a *k*-byte ciphertext C is accomplished by raising C to the RSA private exponent modulo the RSA modulus, and returning the resulting value as a sequence of exactly *k* bytes.

Technically, the above procedures may differ very slightly from certain details of what is specified in X.509.

Executing cryptographic operations using this mechanism can result in the error returns `CKR_DATA_INVALID` (if plaintext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus) and `CKR_ENCRYPTED_DATA_INVALID` (if ciphertext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus).

Constraints on key types and the length of input and output data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table B4-21, X.509 (Raw) RSA: Key and Data Length

Function	Key type	Input length	Output length
`C_Encrypt`[1]	RSA public key	$\leq k$	k
`C_Decrypt`[1]	RSA private key	k	k
`C_Sign`[1]	RSA private key	$\leq k$	k
`C_Verify`[1]	RSA public key	$\leq k$, k (data, followed by signature)	N/A

[1] Single-part operations only.

This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC 9796 block formats.

B4.6.1.10 PKCS #1 v1.5 RSA Signature with Digest

The PKCS #1 v1.5 RSA signature with MD5 mechanism, denoted `CKM_MD5_RSA_PKCS`, performs the same operations described in **[PKCS#1]** with the object identifier `md5WithRSAEncryption`.

Likewise, the PKCS #1 v1.5 RSA signature with SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 mechanisms, denoted `CKM_SHA1_RSA_PKCS`, `CKM_SHA224_RSA_PKCS`, `CKM_SHA256_RSA_PKCS`, `CKM_SHA384_RSA_PKCS`, and `CKM_SHA512_RSA_PKCS` respectively, perform the same operations using the SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 hash functions with the object identifiers `sha1WithRSAEncryption`, `sha224WithRSAEncryption`, `sha256WithRSAEncryption`, `sha384WithRSAEncryption`, and `sha384WithRSAEncryption` respectively.

None of these mechanisms have a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table B4-22: PKCS #1 v1.5 RSA Signatures with Hash: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Sign`	RSA private key	any	k	block type 01
`C_Verify`	RSA public key	any, k (data followed by signature)	N/A	block type 01

The PKCS #1 v1.5 signature scheme requires a minimum size for the size *k* of the modulus, which is 11 bytes plus the size of the DER encoding of a DigestInfo sequence composed of the digest algorithm identifier followed by the value of the digest. Constraints on the size of the modulus are summarized in the following table:

Table B4-23: PKCS #1 v1.5 RSA Signatures with Hash Functions: Modulus Length

Hash	Minimum Modulus Size
MD5	45 bytes
SHA-1	46 bytes
SHA-224	58 bytes
SHA-256	62 bytes
SHA-384	78 bytes
SHA-512	94 bytes

B4.6.1.11 PKCS #1 RSA PSS SIGNATURE WITH DIGEST

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted `CKM_SHA1_RSA_PKCS_PSS`, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in **[PKCS#1]** with the object identifier `id-RSASSA-PSS`, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

The PKCS #1 RSA PSS signature with SHA-224, SHA-256, SHA-384, and SHA-512 mechanisms, denoted `CKM_SHA224_RSA_PKCS_PSS`, `CKM_SHA256_RSA_PKCS_PSS`, `CKM_SHA384_RSA_PKCS_PSS`, `CKM_SHA512_RSA_PKCS_PSS`, perform the same operation using the SHA-224, SHA-256, SHA-384 and SHA-512 hash functions.

The mechanisms have a parameter, a `CK_RSA_PKCS_PSS_PARAMS` structure. The `sLen` field must be less than or equal to *k**-2-*hLen* where *hLen* is the length in bytes of the hash value. *k** is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case *k** is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table B4-24: PKCS #1 RSA PSS Signatures with Hash: Key and Data Length

Function	Key type	Input length	Output length
`C_Sign`	RSA private key	any	*k*
`C_Verify`	RSA public key	any, *k* (data followed by signature)	N/A

Note that the hash algorithm specified by the `hashAlg` field in the `CK_RSA_PKCS_PSS_PARAMS` structure must correspond to the hash algorithm specified in the mechanism identifier.

B4.6.2 DIFFIE-HELLMAN

This section defines the key type "`CKK_DH`" for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of DH key objects.

This section defines the following mechanisms:

```
CKM_DH_PKCS_KEY_PAIR_GEN
CKM_DH_PKCS_DERIVE
```

B4.6.2.1 DIFFIE-HELLMAN PUBLIC KEY OBJECTS

Diffie-Hellman public key objects (object class `CKO_PUBLIC_KEY`, key type `CKK_DH`) hold Diffie-Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table B4-25, Diffie-Hellman Public Key Object Attributes

Attribute	Data type	Meaning
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_PUBLIC_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_DH`
`CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_PRIME`	Big integer	Prime *p*
`CKA_BASE`	Big integer	Base *g*
`CKA_VALUE`	Big integer	Public value *y*

The `CKA_PRIME` and `CKA_BASE` attribute values are collectively the "Diffie-Hellman domain parameters". Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

All attributes are public and can be retrieved using the function `C_GetAttributeValue`. All the three numbers *p*, *g*, *y* must be provided when creating a DH public key object. However, only *p* and *g* must be provided when generating a key pair, because this will generate *y*.

The following is a sample template for creating a Diffie-Hellman public key object:

```
CK_OBJECT_CLASS class    = CKO_PUBLIC_KEY;
CK_KEY_TYPE     keyType = CKK_DH;
CK_BYTE         id[2]    = {0x00, 0x01};
CK_BYTE         prime[] = {...};
CK_BYTE         base[]   = {...};
CK_BYTE         value[] = {...};
CK_BBOOL        true     = CK_TRUE;
CK_ATTRIBUTE    template[] =
{
  {CKA_CLASS,    &class,   sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN,    &true,    sizeof(true)},
  {CKA_ID,       id,       sizeof(id)},
  {CKA_PRIME,    prime,    sizeof(prime)},
  {CKA_BASE,     base,     sizeof(base)},
  {CKA_VALUE,    value,    sizeof(value)}
};
```

B4.6.2.2 DIFFIE-HELLMAN PRIVATE KEY OBJECTS

Diffie-Hellman private key objects (object class `CKO_PRIVATE_KEY`, key type `CKK_DH`) hold Diffie-Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in addition to the common attributes defined for this object class:

Table B4-26, Diffie-Hellman Private Key Object Attributes

Attribute	Data type	Meaning
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_PRIVATE_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_DH`
`CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_DERIVE, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_PRIME`	Big integer	Prime *p*
`CKA_BASE`	Big integer	Base *g*
`CKA_VALUE`	Big integer	Private value *x*
`CKA_VALUE_BITS`	`CK_ULONG`	Length in bits of private value *x*

The `CKA_PRIME` and `CKA_BASE` attribute values are collectively the "Diffie-Hellman domain parameters". Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

The attribute `CKA_VALUE` is protected. Since DH private keys are always sensitive, this attribute can never be retrieved using the function `C_GetAttributeValue`.

When creating a DH private keys, *p*, *g*, and *x* must be provided.

When generating an Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in the private key's template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-Hellman key *pair*, and the Diffie-Hellman parameters for the pair are specified in the template for the Diffie-Hellman public key.

The following is a sample template for creating a Diffie-Hellman private key object:

```
CK_OBJECT_CLASS class   = CKO_PRIVATE_KEY;
CK_KEY_TYPE     keyType = CKK_DH;
CK_BYTE         id[2]   = {0x00, 0x01};
CK_BYTE prime[]   = {...};
CK_BYTE base[]    = {...};
CK_BYTE value[]   = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] =
{
  {CKA_CLASS,     &class,   sizeof(class)},
  {CKA_KEY_TYPE,  &keyType, sizeof(keyType)},
  {CKA_TOKEN,     &true,    sizeof(true)},
  {CKA_ID,        id,       sizeof(id)},
  {CKA_DERIVE,    &true,    sizeof(true)},
  {CKA_PRIME,     prime,    sizeof(prime)},
  {CKA_BASE,      base,     sizeof(base)},
  {CKA_VALUE,     value,    sizeof(value)}
};
```

B4.6.2.3 PKCS #3 Diffie-Hellman Key Pair Generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted `CKM_DH_PKCS_KEY_PAIR_GEN`, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in **[PKCS#3]**. This is what PKCS #3 calls "phase I".

It does not have a parameter.

The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as specified in the `CKA_PRIME` and `CKA_BASE` attributes of the template for the public key. If the `CKA_VALUE_BITS` attribute of the private key is specified, the mechanism limits the length in bits of the private value, as described in PKCS #3. Otherwise, the private value length can be arbitrary.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new public key and the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_PRIME`, `CKA_BASE`, and `CKA_VALUE` (and the `CKA_VALUE_BITS` attribute, if it is not already provided in the template) attributes to the new private key; other attributes required by the Diffie-Hellman public and private key types must be specified in the templates.

B4.6.2.4 PKCS#3 Diffie-Hellman Key Derivation

The PKCS #3 Diffie-Hellman key derivation mechanism, denoted `CKM_DH_PKCS_DERIVE`, is a mechanism for key derivation based on Diffie-Hellman key agreement, as defined in **[PKCS#3]**. This is what PKCS #3 calls "phase II".

It has a parameter, which is the public value of the other party in the key agreement protocol, represented as a Cryptoki "Big integer" (*i.e.*, a sequence of bytes, most-significant byte first).

This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other party. It computes a Diffie-Hellman secret value from the public value and private key according to PKCS #3, and truncates the result according to the `CKA_KEY_TYPE` attribute of the template and, if it has one and the key type supports it, the `CKA_VALUE_LEN` attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the `CKA_VALUE` attribute of the new key; other attributes required by the key type must be specified in the template.

The following restrictions apply for Diffie-Hellman key derivation:

- the derived key can only be of type `CKK_GENERIC_SECRET`. If the attribute `CKA_KEY_TYPE` is not set to `CKK_GENERIC_SECRET`, the function return `CKR_ATTRIBUTE_VALUE_INVALID`;
- it is not possible to truncate the value of the shared secret. The attribute `CKA_VALUE_LEN` must be either omitted or set to the number of bytes necessary to represent the prime *p*.

This mechanism also has the following non-standard rules about the sensitivity of the derived key:

- If the attribute `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY` of the private key is not set or if it is set to its default value of `CK_FALSE`, then the template for the derived key cannot set the value of the attribute `CKA_SENSITIVE` to `CK_FALSE`: it must either not mention the attribute `CKA_SENSITIVE` or set its value to `CK_TRUE`. As a consequence, the value of the derived key can never be retrieved;
- If the attribute `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY` of the private key is set to `CK_TRUE`, then the template for the derived key can set the attribute `CKA_SENSITIVE` to any value. As a consequence, the caller may explicitly choose to allow the derived key to be later retrieved in clear-text using `C_GetAttributeValue`.

B4.6.3 DSA

This section defines the key type "CKK_DSA" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of DSA key objects.

This section defines the following mechanisms:

```
CKM_DSA_KEY_PAIR_GEN
CKM_DSA
CKM_DSA_SHA1
```

B4.6.3.1 DSA PUBLIC KEY OBJECTS

DSA public key objects (object class CKO_PUBLIC_KEY, key type CKK_DSA) hold DSA public keys. The following table defines the DSA public key object attributes, in addition to the common attributes defined for this object class:

Table B4-27, DSA Public Key Object Attributes

Attribute	Data type	Meaning
CKA_CLASS	CK_OBJECT_CLASS	Set to CKO_PUBLIC_KEY
CKA_KEY_TYPE	CK_KEY_TYPE	Set to CKK_DSA
CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_SENSITIVE, CKA_VERIFY, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY	See section B4.4, "*Objects and Attributes*"	
CKA_PRIME	Big integer	Prime *p* (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME	Big integer	Subprime *q* (160 bits)
CKA_BASE	Big integer	Base *g*
CKA_VALUE	Big integer	Public value *y*

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the "DSA domain parameters". See FIPS PUB 186-2 for more information on DSA keys.

All attributes are non-sensitive and can be retrieved using the function C_GetAttributeValue.

When creating a DSA public key object, the public value *y* **and** the domain parameters (*p*, *q*, *g*) must be provided.

When generating a DSA key pair, the domain parameters must be provided in the template of the public key.

The following is a sample template for creating a DSA public key object:

```
CK_OBJECT_CLASS class    = CKO_PUBLIC_KEY;
CK_KEY_TYPE     keyType = CKK_DSA;
CK_BYTE         id[2]    = {0x00, 0x01};
CK_BYTE  prime[]     = {...};
CK_BYTE  subprime[] = {...};
CK_BYTE  base[]      = {...};
CK_BYTE  value[]     = {...};
CK_BBOOL true        = CK_TRUE;
CK_ATTRIBUTE template[] =
{
  {CKA_CLASS,     &class,    sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN,     &true,     sizeof(true)},
  {CKA_ID,        id,        sizeof(id)},
  {CKA_VERIFY,    &true,     sizeof(true)},
  {CKA_PRIME,     prime,     sizeof(prime)},
  {CKA_SUBPRIME, subprime, sizeof(subprime)},
  {CKA_BASE,      base,      sizeof(base)},
  {CKA_VALUE,     value,     sizeof(value)}
};
```

B4.6.3.2 DSA PRIVATE KEY OBJECTS

DSA private key objects (object class `CKO_PRIVATE_KEY`, key type `CKK_DSA`) hold DSA private keys. The following table defines the DSA private key object attributes, in addition to the common attributes defined for this object class:

Table B4-28, DSA Private Key Object Attributes

Attribute	**Data type**	**Meaning**
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_PRIVATE_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_DSA`
`CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_SENSITIVE, CKA_SIGN, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_PRIME`	Big integer	Prime p (512 to 1024 bits, in steps of 64 bits)
`CKA_SUBPRIME`	Big integer	Subprime *q* (160 bits)
`CKA_BASE`	Big integer	Base *g*
`CKA_VALUE`	Big integer	Private value *x*

The `CKA_PRIME`, `CKA_SUBPRIME` and `CKA_BASE` attribute values are collectively the "DSA domain parameters". See FIPS PUB 186-2 for more information on DSA keys.

When creating a DSA private key object, the private value *x* **and** the domain parameters (*p*, *q*, *g*) must be provided.

Note that when generating a DSA private key, the DSA domain parameters are *not* specified in the key's template. This is because DSA private keys are only generated as part of a DSA key *pair*, and the DSA domain parameters for the pair are specified in the template for the DSA public key.

The attribute `CKA_VALUE` is protected. Since DSA private keys can never be non-sensitive, this attribute cannot be retrieved using the function `C_GetAttributeValue`. All other attributes can be retrieved.

The following is a sample template for creating a DSA private key object:

```
  CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
  CK_KEY_TYPE     keyType = CKK_DSA;
  CK_BYTE         id[2]   = {0x00, 0x01};
  CK_BYTE   prime[]    = {...};
  CK_BYTE   subprime[] = {...};
  CK_BYTE   base[]     = {...};
  CK_BYTE   value[]    = {...};
  CK_BBOOL true        = CK_TRUE;
  CK_ATTRIBUTE template[] =
  {
    {CKA_CLASS,      &class,   sizeof(class)},
    {CKA_KEY_TYPE,   &keyType, sizeof(keyType)},
    {CKA_TOKEN,      &true,    sizeof(true)},
    {CKA_ID,         id,       sizeof(id)},
    {CKA_SIGN,       &true,    sizeof(true)},
    {CKA_PRIME,      prime,    sizeof(prime)},
    {CKA_SUBPRIME,   subprime, sizeof(subprime)},
    {CKA_BASE,       base,     sizeof(base)},
    {CKA_VALUE,      value,    sizeof(value)}
  };
```

B4.6.3.3 DSA Key Pair Generation

The DSA key pair generation mechanism, denoted `CKM_DSA_KEY_PAIR_GEN`, is a key pair generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

This mechanism does not have a parameter.

The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as specified in the `CKA_PRIME`, `CKA_SUBPRIME`, and `CKA_BASE` attributes of the template for the public key.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new public key and the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_PRIME`, `CKA_SUBPRIME`, `CKA_BASE`, and `CKA_VALUE` attributes to the new private key. Other attributes supported by the DSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

B4.6.3.4 DSA without hashing

The DSA without hashing mechanism, denoted `CKM_DSA`, is a mechanism for single-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash value.)

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table B4-29, DSA: Key and Data Length

Function	Key type	Input length	Output length
`C_Sign`[1]	DSA private key	20	40
`C_Verify`[1]	DSA public key	20, 40 (data followed by signature)	N/A

B4.6.3.5 DSA with SHA-1

The DSA with SHA-1 mechanism, denoted `CKM_DSA_SHA1`, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. This mechanism computes the entire DSA specification, including the hashing with SHA-1.

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table B4-30, DSA with SHA-1: Key and Data Length

Function	**Key type**	**Input length**	**Output length**
`C_Sign`	DSA private key	any	40
`C_Verify`	DSA public key	any, 40 (data followed by signature)	N/A

B4.6.4 AES

For the Advanced Encryption Standard (AES) see **[FIPS 197]**.

This section defines the key type "`CKK_AES`" for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of key objects.

This section defined the following mechanisms:

```
CKM_AES_KEY_GEN
CKM_AES_ECB
CKM_AES_CBC
CKM_AES_CTR
CKMV_AES_CTR (deprecated)
```

B4.6.4.1 AES SECRET KEY OBJECTS

AES secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_AES`) hold AES keys.

Table B4-31, AES Secret Key Object Attributes

Attribute	Data type	Meaning
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_SECRET_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_AES`
`CKA_TOKEN`, `CKA_ID`, `CKA_PRIVATE`, `CKA_MODIFIABLE`, `CKA_COPYABLE`, `CKA_SENSITIVE`, `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_VALUE`	Byte Array	The value of the key
`CKA_VALUE_LEN`	`CK_ULONG`	Length in bytes of the key. One of: 16, 24, or 32 bytes

When creating an AES secret key object, the attribute `CKA_VALUE` must be specified, but the attribute `CKA_VALUE_LEN` must be omitted.

When generating an AES secret key, the attribute `CKA_VALUE_LEN` must be specified in the template to select the length of the key (16, 24, or 32 bytes).

The attribute `CKA_VALUE` is protected. Unless the key is non-sensitive, it cannot be retrieved using the function `C_GetAttributeValue`.

B4.6.4.2 AES KEY GENERATION

The AES key generation mechanism, denoted `CKM_AES_KEY_GEN`, is a key generation mechanism for NIST's Advanced Encryption Standard.

It does not have a parameter.

The mechanism generates AES keys with a particular length in bytes, as specified in the `CKA_VALUE_LEN` attribute of the template for the key.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

B4.6.4.3 AES-ECB

AES-ECB, denoted `CKM_AES_ECB`, is a mechanism for single- and multiple-part encryption and decryption based on NIST Advanced Encryption Standard and electronic codebook mode.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table B4-32: AES-ECB: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_AES`	multiple of block size	same as input length	no final part
`C_Decrypt`	`CKK_AES`	multiple of block size	same as input length	no final part

B4.6.4.4 AES-CBC

AES-CBC, denoted `CKM_AES_CBC`, is a mechanism for single and multiple-part encryption and decryption based on NIST's Advanced Encryption Standard and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

Constraints on key types and the length of data are summarized in the following table:

Table B4-33: AES-CBC: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_AES`	multiple of block size	same as input length	no final part
`C_Decrypt`	`CKK_AES`	multiple of block size	same as input length	no final part

B4.6.4.5 AES-CTR

AES-CTR, denoted `CKM_AES_CTR`, is a mechanism for a single- and multiple-part encryption and decryption with AES in counter mode.

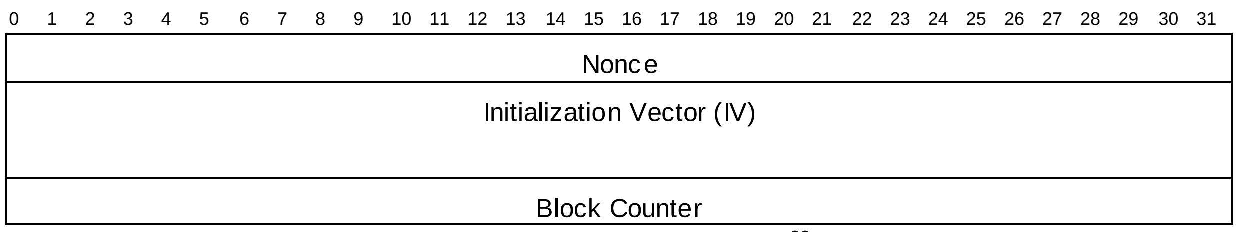
It has a parameter, a `CK_AES_CTR_PARAMS` structure, where the first field indicates the number of bits in the counter block, and the next field is the counter block.

`CK_AES_CTR_PARAMS` is a structure that provides the parameters to the `CKM_AES_CTR` mechanism. It is defined as follows:

```
typedef struct CK_AES_CTR_PARAMS {
   CK_ULONG ulCounterBits;
   CK_BYTE cb[16];
} CK_AES_CTR_PARAMS;
```

The fields of the structure have the following meanings:

- `ulCounterBits`: the number of bits in the counter block (`cb`) that shall be incremented. This number shall be such that $0 <$ `ulCounterBits` ≤ 128. For any values outside this range the mechanism shall return `CKR_MECHANISM_PARAM_INVALID`.
- `cb`: specifies the counter block. It's up to the caller to initialize all the bits in the counter block including the counter bits. The counter bits are the least significant bits of the counter block. They are a big-endian value usually starting with 1. The reset of `cb` is for the nonce, and maybe an optional IV. E.g. as defined in **[RFC3686]**:



This construction permits each packet to consist of up to 2^{32}-1 blocks = 4,294,967,295 blocks = 68,719,476,720 octets.

Generic AES counter mode is described in NIST Special Publication **[NIST800-38A]**, and in **[RFC3686]**. These describe encryption using a counter block which may include a nonce to guarantee uniqueness of the counter block. Since the nonce is not incremented the mechanism parameter must specify the number of counter bits in the counter block.

The block counter is incremented by 1 after each block of plaintext is processed. There is no support for any other increment functions in this mechanism.

If an attempt to encrypt/decrypt is made after an overflow of the counter block's counter bits, then the mechanism shall return `CKR_DATA_LEN_RANGE`. Note that the mechanism should allow the final post increment of the counter to overflow (if it implements it this way) but not allow any further processing after this point. E.g. if `ulCounterBits` = 2 and the counter bits start as 1 then only 3 blocks of data can be processed.

Table B4-34: AES-CTR: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	CKK_AES	multiple of block size	same as input length	no final part
`C_Decrypt`	CKK_AES	multiple of block size	same as input length	no final part

B4.6.4.6 AES-Counter Mode (Non-Standard Mechanism -- Deprecated)

*The mechanism denoted by `CKMV_AES_CTR` is a non-standard mechanism used for AES-CTR. Since the `CKM_AES_CTR` mechanism has been standardized (in Amendment 3 of PKCS11 version 2.20), the non-standard mechanism is now **deprecated**. Its definition is still included in this specification for compatibility reasons.*

The mechanism `CKMV_AES_CTR` has a parameter, which is the initial counter, a 16-byte value coded as a Big Integer. Constraints on key types and the length of data are summarized in the following table:

Table B4-35: AES-CTR: Key and Data Length

Function	*Key type*	*Input length*	*Output length*	*Comments*
`C_Encrypt`	*CKK_AES*	*multiple of block size*	*same as input length*	*no final part*
`C_Decrypt`	*CKK_AES*	*multiple of block size*	*same as input length*	*no final part*

The counter is automatically incremented by 1 at each block. If the counter reaches its maximum value, it is wrapped around to 0.

B4.6.5 DES

This section defines the following mechanisms:

```
CKM_DES_KEY_GEN
CKM_DES_ECB
CKM_DES_CBC
```

B4.6.5.1 DES SECRET KEY OBJECTS

DES secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_DES`) hold single-length DES keys.

Table B4-36, DES Secret Key Object Attributes

Attribute	**Data type**	**Meaning**
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_SECRET_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_DES`
`CKA_TOKEN`, `CKA_ID`, `CKA_PRIVATE`, `CKA_MODIFIABLE`, `CKA_COPYABLE`, `CKA_SENSITIVE`, `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_VALUE`	Byte Array	The value of the key. Must always have its parity bits properly set.
`CKA_VALUE_LEN`	`CK_ULONG`	Length in bytes of the key. Always 8

When creating a DES secret key object, the attribute `CKA_VALUE` must be specified, but the attribute `CKA_VALUE_LEN` must be omitted. The parity bits must be properly set in `CKA_VALUE` as described in **[FIPS 46-3]**.

When generating an DES secret key, the attribute `CKA_VALUE_LEN` can be omitted. If specified, it must be equal to 8.

The attribute `CKA_VALUE` is protected. Unless the key is non-sensitive, it cannot be retrieved using the function `C_GetAttributeValue`.

B4.6.5.2 DES KEY GENERATION

Cipher DES has a key generation mechanism, denoted `CKM_DES_KEY_GEN`.

This mechanism does not have a parameter.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

When DES keys are generated, their parity bits are set properly, as specified in FIPS PUB 46-3. Similarly, when a triple-DES key is generated, each of the DES keys comprising it has its parity bits set properly.

The DES key generator ensures DES keys are neither weak nor semi-weak.

B4.6.5.3 DES ECB

Cipher DES has an electronic codebook mechanism, denoted `CKM_DES_ECB`. It is a mechanism for single- and multiple-part encryption and decryption with DES.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table B4-37: DES ECB: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_DES`	multiple of block size	same as input length	no final part
`C_Decrypt`	`CKK_DES`	multiple of block size	same as input length	no final part

B4.6.5.4 DES CBC

Cipher DES has a cipher-block chaining mode, denoted `CKM_DES_CBC`. It is a mechanism for single- and multiple-part encryption and decryption with DES.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as DES's block size.

Constraints on key types and the length of data are summarized in the following table:

Table B4-38: DES CBC: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_DES`	multiple of block size	same as input length	no final part
`C_Decrypt`	`CKK_DES`	multiple of block size	same as input length	no final part

B4.6.6 Double and Triple-Length DES

This section defines the following mechanisms:

```
CKM_DES2_KEY_GEN
CKM_DES3_KEY_GEN
CKM_DES3_ECB
CKM_DES3_CBC
```

Note that double-length and triple-length DES keys are distinguished by their type (`CKK_DES2` and `CKK_DES3`), but there is a single mechanism for Triple DES (`CKM_DES3_XXX`) operating both with double-length and triple-length keys.

B4.6.6.1 Double-Length DES Key Objects

DES2 secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_DES2`) hold double-length DES keys.

Table B4-39, Double-Length DES Secret Key Object Attributes

Attribute	Data type	Meaning
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_SECRET_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_DES2`
`CKA_TOKEN`, `CKA_ID`, `CKA_PRIVATE`, `CKA_MODIFIABLE`, `CKA_COPYABLE`, `CKA_SENSITIVE`, `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_VALUE`	Byte Array	The value of the key. Must always have its parity bits properly set.
`CKA_VALUE_LEN`	`CK_ULONG`	Length in bytes of the key. Always 16

When creating a DES2 secret key object, the attribute `CKA_VALUE` must be specified, but the attribute `CKA_VALUE_LEN` must be omitted. The parity bits must be properly set in `CKA_VALUE` as described in **[FIPS 46-3]** (*i.e.*, each of the DES keys comprising a DES2 key must have its parity bits properly set).

When generating a DES2 secret key, the attribute `CKA_VALUE_LEN` can be omitted. If specified, it must be equal to 16.

The attribute `CKA_VALUE` is protected. Unless the key is non-sensitive, it cannot be retrieved using the function `C_GetAttributeValue`.

B4.6.6.2 Double-Length DES Key Generation

The double-length DES key generation mechanism, denoted `CKM_DES2_KEY_GEN`, is a key generation mechanism for double-length DES keys. The DES keys making up a double-length DES key both have their parity bits set properly, as specified in **[FIPS 46-3]**.

It does not have a parameter.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key. Other attributes supported by the double-length DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

Double-length DES keys can be used with all the same mechanisms as triple-DES keys: CKM_DES3_ECB and CKM_DES3_CBC. Triple-DES encryption with a double-length DES key is equivalent to encryption with a triple-length DES key with K1=K3 as specified in **[FIPS 46-3]**.

The DES key generator ensures DES keys are neither weak nor semi-weak.

B4.6.6.3 TRIPLE-LENGTH DES KEY OBJECTS

DES3 secret key objects (object class CKO_SECRET_KEY, key type CKK_DES3) hold triple-length DES keys.

Table B4-40, Triple-Length DES Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_CLASS	CK_OBJECT_CLASS	Set to CKO_SECRET_KEY
CKA_KEY_TYPE	CK_KEY_TYPE	Set to CKK_DES3
CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_SENSITIVE, CKA_ENCRYPT, CKA_DECRYPT, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY	See section B4.4, "*Objects and Attributes*"	
CKA_VALUE	Byte Array	The value of the key. Must always have its parity bits properly set.
CKA_VALUE_LEN	CK_ULONG	Length in bytes of the key. Always 24

When creating a DES3 secret key object, the attribute CKA_VALUE must be specified, but the attribute CKA_VALUE_LEN must be omitted. The parity bits must be properly set in CKA_VALUE as described in **[FIPS 46-3]** (*i.e.*, each of the DES keys comprising a DES3 key must have its parity bits properly set).

When generating a DES3 secret key, the attribute CKA_VALUE_LEN can be omitted. If specified, it must be equal to 24.

The attribute CKA_VALUE is protected. Unless the key is non-sensitive, it cannot be retrieved using the function C_GetAttributeValue.

B4.6.6.4 TRIPLE-LENGTH DES KEY GENERATION

The triple-length DES key generation mechanism, denoted CKM_DES3_KEY_GEN, is a key generation mechanism for triple-length DES keys. The DES keys making up a triple-length DES key both have their parity bits set properly, as specified in **[FIPS 46-3]**.

It does not have a parameter.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the double-length DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

The DES key generator ensures DES keys are neither weak nor semi-weak.

B4.6.6.5 TRIPLE DES ORDER OF OPERATIONS

Triple-length DES encryptions are carried out as specified in [FIPS 46-3]: encrypt, decrypt, encrypt. Decryptions are carried out with the opposite three steps: decrypt, encrypt, decrypt. The mathematical representations of the encrypt and decrypt operations are as follows:

DES3-E({K1,K2,K3}, P) = E(K3, D(K2, E(K1, P)))

DES3-D({K1,K2,K3}, C) = D(K1, E(K2, D(K3, P)))

For double-length DES keys, K1=K3.

B4.6.6.6 TRIPLE DES IN ECB MODE

Cipher DES has an electronic codebook mechanism, denoted `CKM_DES3_ECB`. It is a mechanism for single- and multiple-part encryption and decryption with Triple DES.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table B4-41: DES3 ECB: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_DES2`, `CKK_DES3`	multiple of block size	same as input length	no final part
`C_Decrypt`	`CKK_DES2`, `CKK_DES3`	multiple of block size	same as input length	no final part

B4.6.6.7 TRIPLE DES IN CBC MODE

Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the CBC encrypt and decrypt operations are as follows:

DES3-CBC-E({K1,K2,K3}, P) = E(K3, D(K2, E(K1, P + I)))

DES3-CBC-D({K1,K2,K3}, C) = D(K1, E(K2, D(K3, P))) + I

The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the current input block. The addition operation is used is addition modulo-2 (XOR).

Triple DES has a cipher-block chaining mode, denoted `CKM_DES3_CBC`. It is a mechanism for single- and multiple-part encryption and decryption with DES.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as DES's block size.

Constraints on key types and the length of data are summarized in the following table:

Table B4-42: DES3 CBC: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_DES2`, `CKK_DES3`	multiple of block size	same as input length	no final part
`C_Decrypt`	`CKK_DES2`, `CKK_DES3`	multiple of block size	same as input length	no final part

B4.6.7 RC4

This section defines the key type "CKK_RC4" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

This section defines the following mechanisms:

```
CKM_RC4_KEY_GEN
CKM_RC4
```

B4.6.7.1 RC4 SECRET KEY OBJECTS

RC4 secret key objects (object class CKO_SECRET_KEY, key type CKK_RC4) hold RC4 keys. The following table defines the RC4 secret key object attributes:

Table B4-43, RC4 Secret Key Object

Attribute	Data type	Meaning
CKA_CLASS	CK_OBJECT_CLASS	Set to CKO_SECRET_KEY
CKA_KEY_TYPE	CK_KEY_TYPE	Set to CKK_RC4
CKA_TOKEN, CKA_ID, CKA_PRIVATE, CKA_MODIFIABLE, CKA_COPYABLE, CKA_SENSITIVE, CKA_ENCRYPT, CKA_DECRYPT, CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY	See section B4.4, "*Objects and Attributes*"	
CKA_VALUE	Byte array	Key value (1 to 256 bytes)
CKA_VALUE_LEN	CK_ULONG	Length in bytes of key value

When creating an RC4 secret key object, the attribute CKA_VALUE must be specified, but the attribute CKA_VALUE_LEN must be omitted.

When generating an RC4 secret key, the attribute CKA_VALUE_LEN must be specified in the template to select the length of the key (from 1 to 256 bytes).

The attribute CKA_VALUE is protected. Unless the key is non-sensitive, it cannot be retrieved using the function C_GetAttributeValue.

The following is a sample template for creating an RC4 secret key object:

```
CK_OBJECT_CLASS class   = CKO_SECRET_KEY;
CK_KEY_TYPE     keyType = CKK_RC4;
CK_BYTE  value[]  = {...};
CK_BBOOL true     = CK_TRUE;
CK_ATTRIBUTE template[] =
{
  {CKA_CLASS,    &class,   sizeof(class)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_ENCRYPT,  &true,    sizeof(true)},
  {CKA_VALUE,    value,    sizeof(value)}
};
```

B4.6.7.2 RC4 KEY GENERATION

The RC4 key generation mechanism, denoted CKM_RC4_KEY_GEN, is a key generation mechanism for RSA Security's proprietary stream cipher RC4.

It does not have a parameter.

The mechanism generates RC4 keys with a particular length in bytes, as specified in the `CKA_VALUE_LEN` attribute of the template for the key.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key. Other attributes supported by the RC4 key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

B4.6.7.3 RC4 MECHANISM

RC4, denoted `CKM_RC4`, is a mechanism for single- and multiple-part encryption and decryption based on RSA Security's proprietary stream cipher RC4.

It does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table:

Table B4-44, RC4: Key and Data Length

Function	Key type	Input length	Output length	Comments
`C_Encrypt`	`CKK_RC4`	any	same as input length	no final part
`C_Decrypt`	`CKK_RC4`	any	same as input length	no final part

B4.6.8 Generic Secret Key

This section defines the key type "`CKK_GENERIC_SECRET`" for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of key objects.

This section defines the following mechanisms:

`CKM_GENERIC_SECRET_KEY_GEN`

B4.6.8.1 Generic Secret Key Objects

Generic secret key objects (object class **CKO_SECRET_KEY,** key type **CKK_GENERIC_SECRET**) hold generic secret keys. The following table defines the generic secret key object attributes:

Table B4-45, Generic Secret Key Object Attributes

Attribute	**Data type**	**Meaning**
`CKA_CLASS`	`CK_OBJECT_CLASS`	Set to `CKO_SECRET_KEY`
`CKA_KEY_TYPE`	`CK_KEY_TYPE`	Set to `CKK_GENERIC_SECRET`
`CKA_TOKEN`, `CKA_ID`, `CKA_PRIVATE`, `CKA_MODIFIABLE`, `CKA_COPYABLE`, `CKA_SENSITIVE`, `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKA_SIGN`, `CKA_VERIFY`, `CKAV_ALLOW_NON_SENSITIVE_DERIVED_KEY`	See section B4.4, "*Objects and Attributes*"	
`CKA_VALUE`	Byte array	Key value (arbitrary length)
`CKA_VALUE_LEN`	`CK_ULONG`	Length in bytes of key value

Note that generic secrets can be used as keys for HMAC signature and verification operations and for encryption and decryption operations using the implementation-defined block ciphers.

When creating a generic secret key object, the attribute `CKA_VALUE` must be specified, but the attribute `CKA_VALUE_LEN` must be omitted.

When generating a generic secret key, the attribute `CKA_VALUE_LEN` must be specified in the template to select the length of the key.

The attribute `CKA_VALUE` is protected. Unless the key is non-sensitive, it cannot be retrieved using the function `C_GetAttributeValue`.

B4.6.8.2 Generic Secret Key Generation

The generic secret key generation mechanism, denoted `CKM_GENERIC_SECRET_KEY_GEN`, is used to generate generic secret keys. The generated keys take on any attributes provided in the template passed to the `C_GenerateKey` call, and the `CKA_VALUE_LEN` attribute specifies the length in bytes of the key to be generated.

It does not have a parameter.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new key. Other attributes supported by the generic secret key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

B4.6.9 MD5

This section defines the following mechanisms:

```
CKM_MD5
CKM_MD5_HMAC
```

B4.6.9.1 MD5 DIGEST

The MD5 mechanism, denoted `CKM_MD5`, is a mechanism for message digesting, following the MD5 message-digest algorithm defined in **[RFC1321]**.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table B4-46: MD5: Data Length

Function	Data length	Digest length
`C_Digest`	any	16

B4.6.9.2 MD5-HMAC

The MD5 HMAC mechanism, denoted `CKM_MD5_HMAC`, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD5 hash function. The keys it uses are generic secret keys.

The HMAC-MD5 algorithm is specified in FIPS pub 198a **[FIPS 198]**.

This mechanism has no parameter and always produces an output of length 16.

The key used for HMAC signature or verification must:

1. be of length in between 64 and 512 bits
2. have its `CKA_CLASS` attribute set to `CKO_SECRET_KEY`
3. have its `CKA_KEY_TYPE` attribute set to `CKK_GENERIC_SECRET`
4. for a signature operation, have the `CKA_SIGN` attribute set to `CK_TRUE`
5. for a verification operation, have the `CKA_VERIFY` attribute set to `CK_TRUE`.

Table B4-47: MD5-HMAC: Key and Data Length

Function	Key type	Key Length	Input length	Signature length
`C_Sign`	`CKK_GENERIC_SECRET`	$\geq$64 bits, $\leq$512 bits, multiple of 8 bits	any	16 bytes
`C_Verify`				

B4.6.10 SHA-1

This section defines the following mechanisms:

```
CKM_SHA_1
CKM_SHA_1_HMAC
```

B4.6.10.1 SHA-1 DIGEST

The SHA-1 mechanism, denoted `CKM_SHA_1`, is a mechanism for message digesting, following the Secure Hash Algorithm with a 160-bit message digest defined in **[FIPS 180-3]**.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table B4-48: SHA-1: Data Length

Function	Input length	Digest length
`C_Digest`	any	20

B4.6.10.2 SHA-1-HMAC

The SHA-1 HMAC mechanism, denoted `CKM_SHA_1_HMAC`, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys.

The HMAC-SHA-1 algorithm is specified in FIPS pub 198a **[FIPS 198]**.

This mechanism has no parameter and always produces an output of length 20.

The key used for HMAC signature or verification must:

1. be of length in between 80 and 512 bits
2. have its `CKA_CLASS` attribute set to `CKO_SECRET_KEY`
3. have its `CKA_KEY_TYPE` attribute set to `CKK_GENERIC_SECRET`
4. for a signature operation, have the `CKA_SIGN` attribute set to `CK_TRUE`
5. for a verification operation, have the `CKA_VERIFY` attribute set to `CK_TRUE`.

Table B4-49: SHA-1-HMAC: Key and Data Length

Function	Key type	Key Length	Input length	Signature length
`C_Sign`	`CKK_GENERIC_SECRET`	≥80 bits, ≤512 bits, multiple of 8 bits	any	20 bytes
`C_Verify`	`CKK_GENERIC_SECRET`	≥80 bits, ≤512 bits, multiple of 8 bits	any	20 bytes

B4.6.11 SHA-224

This section defines the following mechanisms:

```
CKM_SHA_224
CKM_SHA_224_HMAC
```

B4.6.11.1 SHA-224 DIGEST

The SHA-224 mechanism, denoted `CKM_SHA224`, is a mechanism for message digesting, following the Secure Hash Algorithm with a 224-bit message digest defined in **[FIPS 180-3]**.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table B4-50: SHA-224: Data Length

Function	Input length	Digest length
`C_Digest`	any	28

B4.6.11.2 SHA-224-HMAC

The SHA-224 HMAC mechanism, denoted `CKM_SHA_224_HMAC`, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-224 hash function. The keys it uses are generic secret keys.

The HMAC-SHA-224 algorithm is specified in FIPS pub 198a **[FIPS 198]**.

This mechanism has no parameter and always produces an output of length 28.

The key used for HMAC signature or verification must:

1. be of length in between 112 and 512 bits
2. have its `CKA_CLASS` attribute set to `CKO_SECRET_KEY`
3. have its `CKA_KEY_TYPE` attribute set to `CKK_GENERIC_SECRET`
4. for a signature operation, have the `CKA_SIGN` attribute set to `CK_TRUE`
5. for a verification operation, have the `CKA_VERIFY` attribute set to `CK_TRUE`.

Table B4-51: SHA-224-HMAC: Key and Data Length

Function	Key type	Key Length	Input length	Signature length
`C_Sign`	`CKK_GENERIC_SECRET`	≥112 bits, ≤512 bits, multiple of 8 bits	any	28 bytes
`C_Verify`				

B4.6.12 SHA-256

This section defines the following mechanisms:

```
CKM_SHA_256
CKM_SHA_256_HMAC
```

B4.6.12.1 SHA-256 DIGEST

The SHA-256 mechanism, denoted `CKM_SHA256`, is a mechanism for message digesting, following the Secure Hash Algorithm with a 256-bit message digest defined in **[FIPS 180-3]**.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table B4-52: SHA-256: Data Length

Function	Input length	Digest length
`C_Digest`	any	32

B4.6.12.2 SHA-256-HMAC

The SHA-256 HMAC mechanism, denoted `CKM_SHA256_HMAC`, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-256 hash function. The keys it uses are generic secret keys.

The HMAC-SHA-256 algorithm is specified in FIPS pub 198a **[FIPS 198]**.

This mechanism has no parameter and always produces an output of length 32.

The key used for HMAC signature or verification must:

1. be of length in between 128 and 512 bits
2. have its `CKA_CLASS` attribute set to `CKO_SECRET_KEY`
3. have its `CKA_KEY_TYPE` attribute set to `CKK_GENERIC_SECRET`
4. for a signature operation, have the `CKA_SIGN` attribute set to `CK_TRUE`
5. for a verification operation, have the `CKA_VERIFY` attribute set to `CK_TRUE`.

Table B4-53: SHA-256-HMAC: Key and Data Length

Function	Key type	Key Length	Input length	Signature length
`C_Sign`	`CKK_GENERIC_SECRET`	≥128 bits, ≤512 bits, multiple of 8 bits	any	32 bytes
`C_Verify`				

B4.6.13 SHA-384

This section defines the following mechanisms:

```
CKM_SHA_384
CKM_SHA_384_HMAC
```

B4.6.13.1 SHA-384 DIGEST

The SHA-384 mechanism, denoted `CKM_SHA384`, is a mechanism for message digesting, following the Secure Hash Algorithm with a 384-bit message digest defined in **[FIPS 180-3]**.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table B4-54: SHA-384: Data Length

Function	Input length	Digest length
`C_Digest`	any	48

B4.6.13.2 SHA-384-HMAC

The SHA-384 HMAC mechanism, denoted `CKM_SHA384_HMAC`, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-384 hash function. The keys it uses are generic secret keys.

The HMAC-SHA-384 algorithm is specified in FIPS pub 198a **[FIPS 198]**.

This mechanism has no parameter and always produces an output of length 48.

The key used for HMAC signature or verification must:

1. be of length in between 192 and 1024 bits
2. have its `CKA_CLASS` attribute set to `CKO_SECRET_KEY`
3. have its `CKA_KEY_TYPE` attribute set to `CKK_GENERIC_SECRET`
4. for a signature operation, have the `CKA_SIGN` attribute set to `CK_TRUE`
5. for a verification operation, have the `CKA_VERIFY` attribute set to `CK_TRUE`.

Table B4-55: SHA-384-HMAC: Key and Data Length

Function	Key type	Key Length	Input length	Signature length
`C_Sign`	`CKK_GENERIC_SECRET`	≥192 bits, ≤1024 bits, multiple of 8 bits	any	48 bytes
`C_Verify`				

B4.6.14 SHA-512

This section defines the following mechanisms:

```
CKM_SHA_512
CKM_SHA_512_HMAC
```

B4.6.14.1 SHA-512 DIGEST

The SHA-512 mechanism, denoted `CKM_SHA512`, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in **[FIPS 180-3]**.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table B4-56: SHA-512: Data Length

Function	Input length	Digest length
`C_Digest`	any	64

B4.6.14.2 SHA-512-HMAC

The SHA-512 HMAC mechanism, denoted `CKM_SHA512_HMAC`, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-512 hash function. The keys it uses are generic secret keys.

The HMAC-SHA-512 algorithm is specified in FIPS pub 198a **[FIPS 198]**.

This mechanism has no parameter and always produces an output of length 64.

The key used for HMAC signature or verification must:

1. be of length in between 256 and 1024 bits
2. have its `CKA_CLASS` attribute set to `CKO_SECRET_KEY`
3. have its `CKA_KEY_TYPE` attribute set to `CKK_GENERIC_SECRET`
4. for a signature operation, have the `CKA_SIGN` attribute set to `CK_TRUE`
5. for a verification operation, have the `CKA_VERIFY` attribute set to `CK_TRUE`.

Table B4-57: SHA-256-HMAC: Key and Data Length

Function	Key type	Key Length	Input length	Signature length
`C_Sign`	`CKK_GENERIC_SECRET`	≥256 bits, ≤1024 bits, multiple of 8 bits	any	64 bytes
`C_Verify`				

B4.6.15 Implementation-Defined Block Ciphers

A specific product can define up to 16 implementation-defined block cipher mechanisms. See the documentation of your product for the precise definition of these block ciphers.

This section defines the following mechanisms:

```
CKMV_IMPLEMENTATION_DEFINED_0
CKMV_IMPLEMENTATION_DEFINED_1
CKMV_IMPLEMENTATION_DEFINED_2
CKMV_IMPLEMENTATION_DEFINED_3
CKMV_IMPLEMENTATION_DEFINED_4
CKMV_IMPLEMENTATION_DEFINED_5
CKMV_IMPLEMENTATION_DEFINED_6
CKMV_IMPLEMENTATION_DEFINED_7
CKMV_IMPLEMENTATION_DEFINED_8
CKMV_IMPLEMENTATION_DEFINED_9
CKMV_IMPLEMENTATION_DEFINED_10
CKMV_IMPLEMENTATION_DEFINED_11
CKMV_IMPLEMENTATION_DEFINED_12
CKMV_IMPLEMENTATION_DEFINED_13
CKMV_IMPLEMENTATION_DEFINED_14
CKMV_IMPLEMENTATION_DEFINED_15
```

B4.6.15.1 Operation

Each implementation-defined block cipher mechanism follows the same operation rules:

- The mechanism may be used for single- and multiple- part encryption and decryption only;
- If the mechanism is not supported by the implementation, the functions `C_EncryptInit` and `C_DecryptInit` return `CKR_MECHANISM_INVALID`.
- Some implementation-defined mechanisms may not support multi-part operations. In this case, the function `C_EncryptUpdate` or `C_DecryptUpdate` return `CKR_OPERATION_NOT_INITIALIZED` and the operation state is unaffected
- The mechanism can be used with generic secret keys only
- The mechanism may have a parameter, which is always a byte array. This byte array must be passed in the fields `pParameter` and `ulParameterLen` of the `CK_MECHANISM` structure passed to the `C_EncryptInit` and `C_DecryptInit` functions. The content and interpretation of this byte array depends on the mechanism and is therefore implementation-defined
- For some mechanisms, the input data has certain length constraints, because the mechanism's input data must consist of an integral number of blocks. If these constraints are not satisfied, then the functions `C_Encrypt`, `C_EncryptUpdate`, `C_EncryptFinal`, `C_Decrypt`, `C_C_DecryptUpdate`, and `C_DecryptFinal` return the error code `CKR_DATA_LEN_RANGE`.

Chapter B5 EXTERNAL SECURE STORAGE API

This chapter is the specification of the External Secure Storage API.

B5.1 About the External Secure Storage API

B5.1.1 Objectives

This chapter defines the API for accessing a Secure Storage service.

With more and more applications storing sensitive data, there is a need for a secure storage service that manages the security of persistent data in a transparent way. For example, DRM agents can use the Secure Storage service for storing cryptographic keys, rights or certificates.

By using the Secure Storage service, applications can maximize the benefits from the security features provided by the hardware platform:

- The Secure Storage service uses cryptography to guarantee the authenticity, integrity and confidentiality of persistent data. The cryptographic keys used to reach this security-level are managed and accessed exclusively by the implementation.
- When the device provides a hardware cryptographic key (i.e. a key that is stored on-chip), the secure storage service can enhance the security-level of sensitive data by deriving keys from that unique key and sealing persistent data. The persistent data is thus bound to the device so that it becomes very hard to forge or to clone a device.
- When a hardware component of the device implements a monotonic counter, the Secure Storage service enables strong protection against replay attacks.
- The actual storage of the data may use a dedicated persistent storage facility or may rely on the operating system file-system. Because keys or counters are managed within the implementation itself, the overall protection of the storage is still guaranteed. This protection is applied globally to the whole storage space, so corruption of any part of the physical storage results in a global invalidation of the storage space. The way to recover from such a situation is implementation defined and may require reformatting storage space.

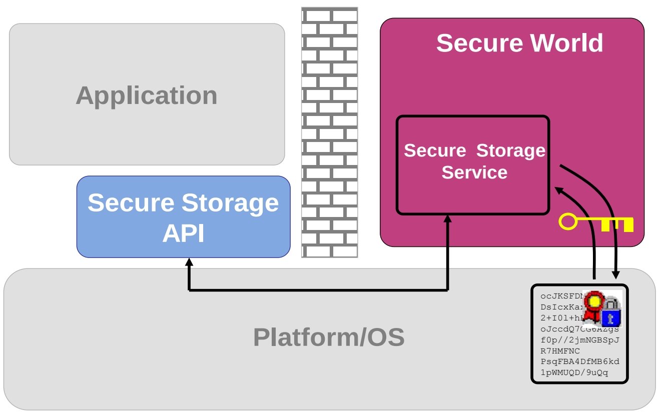


Figure B5-1: Secure Storage API and Secure Storage Service.

B5.1.2 Definition of Clients

Internally, the External Secure Storage API makes use of the `TEEC_LOGIN_AUTHENTICATION` and `TEEC_LOGIN_USER_APPLICATION` login types defined in section B3.3.5.6:

- If the API can locate a signature file associated with the calling application (function `TEEC_ReadSignatureFile`, see section B3.3.5.16), then the login type `TEEC_LOGIN_AUTHENTICATION` is used and the identity of the calling application is determined by the UUID in its manifest
- If no signature file is found, the client is identified using the login type `TEEC_LOGIN_USER_APPLICATION`. This involves OS-specific mechanisms.

The method of login and the client identity fully determines the storage space that an application accesses. In particular, this means that authenticated applications and identified applications access completely separated storage spaces. Furthermore, authenticated applications with different identities are associated to different virtual storage spaces and identified applications with different identities are associated to different virtual storage spaces.

The client login is performed when the application calls the function `SSTInit`; if a signature file is found but the authentication fails for any reason, the error code `SST_ERROR_GENERIC` is returned.

B5.1.3 CHARACTERISTICS OF THE API

Applications can access the Secure Storage service through the Secure Storage API.

The Secure Storage API has the following characteristics:

- Each Client has an associated storage. A Client has exclusive, private access to its storage.
- The Secure Storage API defines a flat file structure. Clients must consider that each file is at the same level within the file system.
- The Secure Storage API is thread-safe. That is, it can be called from multiple concurrent threads.
- The Secure Storage provides confidentiality, integrity, device binding, and atomicity as decribed in section B2.2.10
- Filenames are encoded as a zero-terminated string of bytes. The length of the string shall not exceed `SST_MAX_FILENAME` bytes (excluding the terminating zero character). Filenames shall not contain the following characters:
    - Control characters: char < 0x20
    - Separator character '/': char = 0x2F
    - Separator character '\': char = 0x5C
    - Control characters: char ≥ 0x7F
- Each handle opened on a file has an associated file position indicator. When the file is opened, the position indicator is initially set at the beginning of the file. The file position indicator may be set beyond the existing data in the file. The file position indicator has a maximum value which is `SST_MAX_FILE_POSITION`.

B5.2 External Secure Storage API Reference

This section lists all the types, error codes and functions of the API.

B5.2.1 Types

B5.2.1.1 Basic types

See section B1.1, "*Common Types*" for a definition of the basic types used in this specification.

B5.2.1.2 SST_ERROR

```
typedef uint32_t SST_ERROR;
```

The error type `SST_ERROR` for the Secure Storage service. An alias for `S_RESULT`.

B5.2.1.3 SST_HANDLE

```
typedef uint32_t SST_HANDLE;
```

The file handle type `SST_HANDLE` for the Secure Storage service. The values taken depend on the implementation and should be considered opaque. An alias for `S_HANDLE`.

B5.2.1.4 SST_FILE_INFO

```
typedef struct SST_FILE_INFO
{
   char*    pName;
   uint32_t nSize;
} SST_FILE_INFO;
```

The `SST_FILE_INFO` structure contains information about a file.

B5.2.1.5 SST_WHENCE

```
typedef enum
{
   SST_SEEK_SET,
   SST_SEEK_CUR,
   SST_SEEK_END
} SST_WHENCE;
```

The `SST_WHENCE` enumeration for defining from where to start calculating a new file position.

B5.2.2 Error Codes

See the section B1.2, "*Common Error codes*" for a definition of the values of the error codes described here.

SST_ERROR_ACCESS_CONFLICT

A permissions conflict is detected while trying to open the file, or, when trying to rename the file, a file with the same file name already exists.

SST_ERROR_ACCESS_DENIED

The access to the file is denied. The file handle does not have the appropriate permission to perform the operation requested.

SST_ERROR_BAD_PARAMETERS

One or more of the parameters used is invalid.

SST_ERROR_CORRUPTED

The file exists but its content is corrupted. This may be due, for example, to a physical error on the storage device.

SST_ERROR_ITEM_NOT_FOUND

The specified file was not found on the file system.

SST_ERROR_GENERIC

A generic error (for example a communication error with the service, other than an out of memory error).

SST_ERROR_NO_SPACE

The persistent storage space is full. No further data may be added.

SST_ERROR_OUT_OF_MEMORY

Insufficient memory available to complete the current operation.

SST_ERROR_OVERFLOW

The value of the file position indicator resulting from the `SSTSeek` operation would be greater than `SST_MAX_FILE_POSITION`.

SST_SUCCESS

Successful operation.

B5.2.3 Other Constants

SST_ NULL_HANDLE

Constant value for a null handle.

SST_MAX_FILENAME

Constant value for the maximum length of a filename.

SST_MAX_FILE_POSITION

The maximum value for the file position indicator.

SST_O_READ

Constant value flag for read access permission.

SST_O_WRITE

Constant value flag for write access permission.

SST_O_WRITE_META

Constant value flag for write-meta access permission.

SST_O_CREATE

Constant value flag for create access permission.

SST_O_EXCLUSIVE

Constant value flag for exclusive access permission.

SST_O_SHARE_READ

Constant value flag for shared read access permission.

SST_O_SHARE_WRITE

Constant value flag for shared write access permission.

The following table provides the value of all the constants defined in this specification:

Constant Name	Value
SST_NULL_HANDLE	0x00000000
SST_MAX_FILENAME	0x00000040
SST_MAX_FILE_POSITION	0xFFFFFFFF
SST_O_READ	0x00000001
SST_O_WRITE	0x00000002
SST_O_WRITE_META	0x00000004
SST_O_CREATE	0x00000200
SST_O_EXCLUSIVE	0x00000400
SST_O_SHARE_READ	0x00000010
SST_O_SHARE_WRITE	0x00000020

B5.2.4 Functions

Note

The IN and OUT keywords are used in the function declarations to tag the pointer parameters:

- The IN tag means the pointer is an input parameter,
- The OUT tag means the pointer is an output parameter,

The IN OUT tags mean the pointer is an input and output parameter.

B5.2.4.1 SSTInit

```
SST_ERROR SSTInit(void)
```

This function initializes the Secure Storage service.

This function needs to be called by each Client that uses the Secure Storage service, and before any other calls to the Secure Storage service are made. If any function is called before `SSTInit` has been called, unless otherwise specified, that function will return the error code `SST_ERROR_GENERIC`.

If the Secure Storage service is already initialized, this function does nothing and returns `SST_SUCCESS`.

Parameters

None.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other error.

B5.2.4.2 SSTTerminate

```
SST_ERROR SSTTerminate(void)
```

This function terminates the Secure Storage service.

Before the Secure Storage service terminates, all pending operations are terminated and all the files that are currently opened are closed.

If the Secure Storage service is not initialized, this function does nothing and returns `SST_SUCCESS`.

Depending on the implementation, this function may block until all pending operations complete.

Parameters

None.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_GENERIC`: For any error.

B5.2.4.3 SSTOPEN

```
SST_ERROR SSTOpen(
       IN  const char* pFilename,
           uint32_t    nFlags,
           uint32_t    nReserved,
       OUT SST_HANDLE* phFile);
```

The function `SSTOpen` creates or opens a file referred to by `pFilename`. It returns a handle that can be used to access the file.

A created file is unique to the Client making the call to `SSTOpen` and cannot be accessed by other Clients. Any attempt made to open a file with the same name from a different Client will open a file distinctly separated by the underlying Secure Storage service.

Each handle opened through the `SSTOpen` function has an associated file position indicator. When the file is opened, the position indicator is initially set at the beginning of the file.

The `nFlags` parameter controls the access privileges and file modification behavior of the file to be opened. It is constructed by a bitwise inclusive OR of flags from the following list:

- `SST_O_READ`: the file is opened with read access rights. This allows calling the function `SSTRead`
- `SST_O_WRITE`: the file is opened with write access rights. This allows calling the function `SSTWrite`
- `SST_O_WRITE_META`: the file is opened with write-meta access rights. This allows calling the functions `SSTCloseAndDelete` and `SSTRename`
- `SST_O_SHARE_READ`: the caller allows another handle on the file to be created with read access
- `SST_O_SHARE_WRITE`: the caller allows another handle on the file to be created with write access
- `SST_O_CREATE`: if the file does not exist, it is created. If it exists, it is merely opened.
- `SST_O_EXCLUSIVE`: this flag is meaningful only if `SST_O_CREATE` is set and is ignored otherwise. When `SST_O_CREATE` and `SST_O_EXCLUSIVE` are both set, the file is created only if it does not already exist. Otherwise, the error `SST_ERROR_ACCESS_CONFLICT` is returned.

Any bits in `nFlags` not defined above are reserved for future use and must be set to zero. Otherwise, the error code `SST_ERROR_BAD_PARAMETERS` is returned.

Multiple handles may be opened on the same file, but sharing must be explicitly allowed. More precisely, at any one time, the following constraints apply: if more than one handle is open on the same file, and if any of these handles were opened with the flag `SST_O_READ`, then all file handles must have been opened with the flag `SST_O_SHARE_READ`.

There is similar constraint with the flags `SST_O_WRITE` and `SST_O_SHARE_WRITE`.

The write-meta access right is not shareable. At any one time, there can be only one handle opened on a file when that handle has the `SST_O_WRITE_META` flag set - this is true regardless of the `SST_O_SHARE_READ` or `SST_O_SHARE_WRITE` flags being set.

If `SSTOpen` is called in violation of these constraints the function returns the error code `SST_ERROR_ACCESS_CONFLICT`.

Practically, opening a file with the `SST_O_READ` flag set results in a file handle with read access rights. Opening a file with the `SST_O_SHARE_READ` flag set allows other handles to be opened with read access. Omitting this flag locks the file for reading by any other handle. Opening a file with no read access flags results in locking the file against reading, both by the current handle and any subsequent handles.

Examples

The following examples illustrate the behavior of the `SSTOpen` function when called twice on the same file. Note that for readability, the flag names used in the table have been abbreviated by removing the '`SST_O_`' prefix from their name, and any non-`SST_SUCCESS` error codes have been shortened by removing the '`SST_ERROR_`' prefix.

Value of nFlags for SFileOpen 1	Value of nFlags for SFileOpen 2	Return Code of SFileOpen 2	Comments
`READ`	`READ`	`ACCESS_CONFLICT`	The file handles have not been opened with the flag `SHARE_READ`. Only the first call will succeed.
`READ | SHARE_READ`	`READ`	`ACCESS_CONFLICT`	Not all the file handles have been opened with the flag `SHARE_READ`. Only the first call will succeed.
`READ | SHARE_READ`	`READ | SHARE_READ`	`SST_SUCCESS`	All the file handles have been opened with the flag `SHARE_READ`.
`READ`	`WRITE`	`ACCESS_CONFLICT`	Files handles are not opened with share flags. Only the first call will succeed.
`READ | SHARE_READ | SHARE_WRITE`	`WRITE | SHARE_READ | SHARE_WRITE`	`SST_SUCCESS`	All the file handles have been opened with the share flags.
`READ | SHARE_READ | WRITE | SHARE_WRITE`	`WRITE_META`	`ACCESS_CONFLICT`	The write-meta flag indicates an exclusive access to the file. Only the first call will succeed.
`SHARE_READ`	`WRITE | SHARE_WRITE`	`ACCESS_CONFLICT`	A file can be opened with only share flags, which can be used to lock the access to a file against a given mode (here we prevent subsequent accesses in write mode).
`0`	`READ | SHARE_READ`	`ACCESS_CONFLICT`	A file can be opened with no flag set, which can be used to completely lock all subsequent attempts to access the file. Only the first call will succeed.

Parameters

- `pFilename`: The name of the file encoded as a zero-terminated string of bytes. The maximum length of the string and the set of characters allowed are specified in section B5.1.2. If the length of the string is too long or if the string contains invalid characters or if this pointer is `NULL`, the error `SST_ERROR_BAD_PARAMETERS`is returned.
- **`nFlags`**: The flags under which the file is opened. If the value is not 0 or a combination of valid flags, the error `SST_ERROR_BAD_PARAMETERSis returned.`

- **`nReserved`**: Reserved for future use. Should be set to zero.
- `phFile`: A pointer to the variable, which, upon successful completion, contains the opened handle. Upon failure, the value pointed to by `phFile` is set to `SST_NULL_HANDLE`. If this pointer is `NULL`, the error `SST_ERROR_BAD_PARAMETERS` is returned.

Return Value

- `SST_SUCCESS:` On successful completion of this function
- `SST_ERROR_ACCESS_CONFLICT`: An access right conflict was detected while opening the file.
- `SST_ERROR_BAD_PARAMETERS`: A parameter is invalid: `pFilename` is `NULL`, or the file name is too long or contains invalid characters, or the value of `nFlags` is not 0 or a combination of valid flags, or `phFile` is `NULL.`
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_ITEM_NOT_FOUND`: The specified file doesn't exist in the specified storage.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.4 SSTCloseHandle

```
SST_ERROR SSTCloseHandle(SST_HANDLE hFile)
```

This function closes a file handle. Upon successful completion, the file handle is no longer valid.

If `hFile` is `SST_NULL_HANDLE`, the function does nothing and returns `SST_SUCCESS`.

Parameters

- `hFile`: The file handle.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: `hFile` is an invalid file handle (but not `SST_NULL_HANDLE`).
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.5 SSTREAD

```
SST_ERROR SSTRead(
           SST_HANDLE hFile,
       OUT uint8_t*   pBuffer,
           uint32_t   nSize,
       OUT uint32_t*  pnCount)
```

This function attempts to read nSize bytes from the file associated with the handle hFile into the buffer pointed to by pBuffer.

The file handle must have been opened with the SST_O_READ flag.

pBuffer points to a memory area that must have already been allocated by the user, which must be of at least nSize bytes in length, and must not be NULL.

The bytes are read starting at the position in the file associated with the file handle. The handle's file position indicator is incremented by the number of bytes actually read.

On completion, SSTRead sets *pnCount to the number of bytes actually read.

No data transfer occurs past the current end-of-file. If an attempt is made to read past the end-of-file, the SSTRead function stops reading data at the end-of-file and returns the data read until the end-of-file. The file position indicator is then set at the end-of-file.

The value written to *pnCount may be less than nSize if the number of bytes until the end-of-file is less than nSize. This is the only case where *pnCount may be less than nSize.

If the file position indicator is initially at the end-of-file or beyond, the file position indicator is not changed, no bytes are copied to *pBuffer and *pnCount is set to 0.

Parameters

- hFile: The file handle.
- pBuffer: A pointer to the memory which, upon successful completion, contains the bytes read. If it is NULL, the error SST_ERROR_BAD_PARAMETERS is returned.
- nSize: The number of bytes to read.
- pnCount: A pointer to the variable which upon successful completion contains the number of bytes read. If it is NULL, the error SST_ERROR_BAD_PARAMETERS is returned.

Return Value

- SST_SUCCESS: On successful completion of this function
- SST_ERROR_ACCESS_DENIED: The file handle has not been opened with the read access right.
- SST_ERROR_BAD_PARAMETERS: A parameter is invalid: hFile is not a valid file handle or pBuffer is NULL or pnCount is NULL.
- SST_ERROR_CORRUPTED: Data corruption is detected.
- SST_ERROR_OUT_OF_MEMORY: If there is not enough memory left to complete the operation.
- SST_ERROR_GENERIC: For any other reason.

B5.2.4.6 SSTWRITE

```
SST_ERROR SSTWrite (
           SST_HANDLE     hFile,
        IN const uint8_t* pBuffer,
           uint32_t       nSize)
```

The `SSTWrite` function writes `nSize` bytes from the buffer pointed to by `pBuffer` to the file associated with the open file handle `hFile`.

If the current file position indicator points before the end-of-file, then `nSize` bytes are written to the file. If the current file position indicator points beyond the end-of-file, then the file is first extended with zero bytes until the file length matches the file position indicator, and then `nSize` bytes are written to the file.

The file position indicator for the file is advanced by `nSize`. The file position indicator of other file handles opened on the same file is not changed.

The size of the file can be increased as a result of this operation.

Writing in a file is atomic: either the operation completes successfully, or no write is done at all. This is true even if the current file position indicator is beyond the end of the file.

The file handle must have been opened with the `SST_O_WRITE` flag.

Parameters

- `hFile`: The file handle.
- `pBuffer`: The buffer containing the data to be written.
- `nSize`: The number of bytes to write

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_ACCESS_DENIED`: The file handle has not been opened with write access.
- `SST_ERROR_BAD_PARAMETERS`: A parameter is invalid: `hFile` is not a valid file handle or `pBuffer` is `NULL`.
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_NO_SPACE`: Insufficient storage space is available.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.7 SSTTruncate

```
SST_ERROR SSTTruncate(
           SST_HANDLE hFile,
           uint32_t   nSize)
```

The function `SSTTruncate` changes the size of a file. If `nSize` is less than the current size of the file, all bytes beyond `nSize` are removed and the size of the file is set to `nSize`. If `nSize` is greater than the current size of the file, then the file is extended by adding zero bytes at the end of the file.

Truncating a file is atomic: either the file is successfully truncated or nothing happens.

The file handle must have been opened with the `SST_O_WRITE` flag.

This operation does not change the file position indicator of any handle opened on the file. Note that if the current file position indicator of such a handle is beyond `nSize`, the file position indicator will point beyond the end-of-file indicator after truncation.

Parameters

- `hFile`: The file handle.
- `nSize`: The new size of the file

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_ACCESS_DENIED`: The file handle has not been opened with the write access right.
- `SST_ERROR_BAD_PARAMETERS`: `hFile` is not a valid file handle.
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_NO_SPACE`: Insufficient storage space is available.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.8 SSTSeek

```
SST_ERROR SSTSeek(
           SST_HANDLE hFile,
           int32_t    nOffset,
           SST_WHENCE eWhence)
```

The `SSTSeek` function sets the file position indicator associated with the file handle.

The parameter `eWhence` controls the meaning of `nOffset`:

- If `eWhence` is `SST_SEEK_SET`, the file position indicator is set to `nOffset` bytes from the beginning of the file.
- If `eWhence` is `SST_SEEK_CUR`, the file position indicator is set to its current location plus `nOffset`.
- If `eWhence` is `SST_SEEK_END`, the file position indicator is set to the size of the file plus `nOffset`.

The `SSTSeek` function may be used to set the file position indicator beyond the existing data in the file. This does not constitute an error. However, the file position indicator has a maximum value which is `SST_MAX_FILE_POSITION`. If the value of the file position indicator resulting from this operation would be greater than `SST_MAX_FILE_POSITION`, the error `SST_ERROR_OVERFLOW` is returned and the file position is left unchanged.

If an attempt is made to move the file position indicator before the beginning of the file, the file position indicator is set at the beginning of the file. This does not constitute an error.

Parameters

- `hFile`: The file handle.
- `nOffset`: The number of bytes to move the file position indicator. A positive value moves the file position indicator forward; a negative value moves the file position indicator backward.
- `eWhence`: Selects the position in the file from which to calculate the new position.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: `hFile` is not a valid handle or if `eWhence` is not `SST_SEEK_SET`, `SST_SEEK_CUR` or `SST_SEEK_END`.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_OVERFLOW`: If the value of the file position indicator resulting from this operation would be greater than `SST_MAX_FILE_POSITION`.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.9 SSTTELL

```
SST_ERROR SSTTell (
           SST_HANDLE hFile,
       OUT uint32_t*  pnPos)
```

The `SSTTell` function obtains the current value of the file position indicator for the specified file handle.

Parameters

- `hFile`: The file handle.
- `pnPos`: A pointer to the variable which, upon successful completion, contains the current position in the file. The current position is expressed as the number of bytes from the beginning of the file. The variable is set to zero upon failure.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: One of the parameters is invalid: `hFile` is not a valid handle or `pnPos` is `NULL`.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.10 SSTEof

```
SST_ERROR SSTEof(
           SST_HANDLE hFile,
       OUT bool*      pbEof)
```

The `SSTEof` function determines if the current file position indicator is at the end of the file.

Parameters

- `hFile`: The file handle.
- `pbEof`: A pointer to the variable which, upon successful completion, and if the current file position indicator for `hFile` is at the end of the file or beyond, contains `true`. Otherwise, `false` is written.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: One of the parameters is invalid: `hFile` is not a valid file handle or `pbEof` is `NULL`.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.11 SSTCloseAndDelete

```
SST_ERROR SSTCloseAndDelete(SST_HANDLE hFile)
```

The `SSTCloseAndDelete` function deletes a file. After this function returns successfully, the file handle is closed and can no longer be used.

The file handle must have been opened with the write-meta access right.

If `hFile` is `SST_NULL_HANDLE`, the function does nothing and returns `SST_SUCCESS`.

Deleting a file is an atomic operation: either the file is deleted or nothing happens.

Parameters

- `hFile`: The file handle.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_ACCESS_DENIED`: The file handle has not been opened with the write-meta access right.
- `SST_ERROR_BAD_PARAMETERS`: `hFile` is not a valid file handle (but not the `SST_NULL_HANDLE` handle).
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.12 SSTRename

```
SST_ERROR SSTRename(
              SST_HANDLE  hFile,
          IN const char* pNewFilename);
```

The function `SSTRename` changes the name of a file.

The file handle must have been opened with the write-meta access right.

Renaming a file is an atomic operation: either the file is renamed or nothing happens.

Parameters

- `hFile`: The file handle.
- `pNewFilename`: The new file name, encoded as a zero-terminated string of bytes. The maximum length of the string and the set of characters allowed are specified in section B5.1.2.

Return Value

- `SST_SUCCESS:` On successful completion of this function
- `SST_ERROR_ACCESS_CONFLICT`: If a file with the same name already exists.
- `SST_ERROR_ACCESS_DENIED`: The file handle has not been opened with the write-meta access right.
- `SST_ERROR_BAD_PARAMETERS`: `hFile` is not a valid file handle or `pNewFilename` is `NULL` or `pNewFilename` is too long or contains invalid characters
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_NO_SPACE`: Insufficient storage space is available.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.13 SSTGETSIZE

```
SST_ERROR SSTGetSize(
       IN  const char* pFilename,
       OUT uint32_t*   pnFileSize)
```

The `SSTGetSize` function returns the current size of a file. The file is referenced by its name. This function succeeds even if file handles are currently opened on the file.

Parameters

- `pFilename`: The name of the file, encoded as a zero-terminated string of bytes. The maximum length of the string and the set of characters allowed are specified in section B5.1.2.
- `pnFileSize`: A pointer to the variable which, upon successful completion, contains the file size in bytes.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: One of the parameters is invalid: `pFilename` is `NULL` or `pFilename` is too long or contains invalid characters or `pnFileSize` is `NULL`.
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_ITEM_NOT_FOUND`: The specified file was not found in the specified storage.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.14 SSTENUMERATIONSTART

```
SST_ERROR SSTEnumerationStart (
       IN  const char* pFilenamePattern,
           uint32_t    nReserved1,
           uint32_t    nReserved2,
       OUT SST_HANDLE* phFileEnumeration);
```

The function `SSTEnumerationStart` starts an enumeration of a set of files. A set of files to be enumerated is defined as all the files whose name matches a given pattern. A pattern can contain the wildcards '*' and '?' which have the following meanings:

- '*' is the wildcard for 0 or more characters.
- '?' is the wildcard for 1 character.

In order to specify '*' and '?' characters in the pattern string (not interpreted as wildcards), these characters must be preceded by the escape '\' character. A '\' character that is not directly followed by '*' or '?' is forbidden, and will cause the return of an `SST_ERROR_BAD_PARAMETERS` error code.

The files are enumerated using the function `SSTEnumerationGetNext`.

The enumeration does not necessarily reflect a given consistent state of the storage: during the enumeration, other threads may create, delete, or rename files.

Once no longer required, the handle must be closed by calling `SSTEnumerationCloseHandle`.

Parameters

- `pFilenamePattern`: The filename pattern – a zero-terminated string of bytes. The pattern shall not contain the following characters:
    - Control characters: char < 0x20
    - Separator character '/': char = 0x2F
    - Separator character '\' (char = 0x5C) not directly followed by asterisk character '*' (char = 0x2A) or question-mark character '?' (char = 0x3F)
    - Control character: char ≥ 0x7F

This parameter may be `NULL`. In this case, all the files in the storage-space of the Client are enumerated.

- nReserved1, `nReserved2`: Reserved for future use. Should be set to zero.
- `phFileEnumeration`: A pointer filled with a handle on the file enumeration. Set to `SST_NULL_HANDLE` on error.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: `phFileEnumeration` is `NULL`.
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.15 SSTEnumerationCloseHandle

```
SST_ERROR SSTEnumerationCloseHandle(
             SST_HANDLE hFileEnumeration);
```

The function `SSTEnumerationCloseHandle` closes an enumeration handle and frees the internal resources used by the enumeration.

If `hFileEnumeration` is set to `SST_NULL_HANDLE`, this function does nothing and returns `SST_SUCCESS`.

Parameters

- `hFileEnumeration`: A handle on the file enumeration.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: If `hFileEnumeration` is an invalid handle (but not `SST_NULL_HANDLE`).
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.16 SSTEnumerationGetNext

```
SST_ERROR SSTEnumerationGetNext (
                SST_HANDLE       hFileEnumeration,
          OUT SST_FILE_INFO** ppFileInfo);
```

The function `SSTEnumerationGetNext` retrieves the information of the next file in the file enumeration.

If there are no more elements in the file enumeration, this function returns the error code `SST_ERROR_ITEM_NOT_FOUND`.

On error, `*ppFileInfo` is set to `NULL`.

Parameters

- `hFileEnumeration`: A handle on the file enumeration.
- `ppFileInfo`: A pointer filled with a pointer to a structure `SST_FILE_INFO` allocated by the API. This structure must be deallocated using `SSTDestroyFileInfo`. This will deallocate the structure as well as the string it references. Note that `SSTDestroyFileInfo` must be called before `SSTTerminate`, otherwise, the memory used for the structure will leak.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_BAD_PARAMETERS`: One of the parameters is invalid: `hFileEnumeration` is an invalid handle or `ppFileInfo` is `NULL`.
- `SST_ERROR_CORRUPTED`: Data corruption is detected.
- `SST_ERROR_ITEM_NOT_FOUND`: There is no more elements in the file enumeration.
- `SST_ERROR_OUT_OF_MEMORY`: If there is not enough memory left to complete the operation.
- `SST_ERROR_GENERIC`: For any other reason.

B5.2.4.17 SSTDestroyFileInfo

```
SST_ERROR SSTDestroyFileInfo(IN SST_FILE_INFO* pFileInfo);
```

Destroys a file info structure allocated by the implementation. This also destroys the string referenced in the structure.

If `pFileInfo` is set to `NULL`, then the function does nothing is returns `SST_SUCCESS`.

There is no way to deallocate a file information structure after `SSTTerminate` is called and the function `SSTDestroyFilInfo` will return `SST_ERROR_GENERIC` in this case. So, all the file information structures must be freed before the library is terminated, or a memory leak will occur.

Parameters

- `pFileInfo`: A pointer to a structure `SST_FILE_INFO` allocated by a call to `SSTEnumerationGetNext`.

Return Value

- `SST_SUCCESS`: On successful completion of this function
- `SST_ERROR_GENERIC`: For any error.

Chapter B6 EXTERNAL MONOTONIC COUNTER API

This chapter is the specification of the External Monotonic Counter API.

B6.1 About the External Monotonic Counter API

This chapter is the specification of the External Monotonic Counter API of the Trusted Foundations. A Monotonic Counter is a persistent register whose value can only be increased over time. It is useful to generate unique serial numbers or, in general, as a means to build counter-measures against replay attacks.

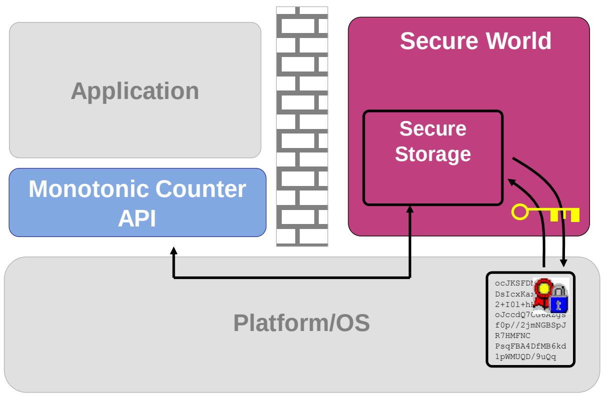


Figure B6-1 Monotonic Counter API

This API provides access to monotonic counters implemented within the Trusted Foundations and used by external normal world Clients.

These counters therefore leverages the full protection provided by the Trusted Foundations. In particular, the monotonicity of the counter is guaranteed by the Secure World and may rely on secure hardware.

In this API, a Monotonic Counter has the following characteristics:

- Its value is a 64-bit integer. The full range is available and the counter can therefore never realistically overflow;
- It is initialized to a random value
- It can be retrieved at any time using the function `SMonotonicCounterGet`;
- It can be incremented using the function `SMonotonicCounterIncrement`. The increment is always exactly one. The increment operation is atomic: if the power goes down during the operation, the counter is either unchanged or incremented by one, but nothing in between

The API is designed to provide access to multiple monotonic counters, although, in this version, there is only a single counter shared between all clients.

In general, this API is thread-safe, unless stated otherwise: multiple threads may access the counters and the accesses are serialized.

B6.2 External Monotonic Counter API Reference

B6.2.1 Types

B6.2.1.1 Basic Types

See section B1.1, “*Common Types*” for a definition of the basic types used in this specification

B6.2.1.2 S_MONOTONIC_COUNTER_VALUE

```
typedef struct
{
   uint32_t nLow;
   uint32_t nHigh;
}
S_MONOTONIC_COUNTER_VALUE;
```

This type denotes the value of a monotonic counter. This value is 64-bit integer expressed on two 32-bit words:

- nLow contains the least-significant 32 bits of the value
- nHigh contains the most-significant 32 bits of the value

B6.2.2 Constants

B6.2.2.1 Error Codes

See section B1.2, “*Common Error codes*” for a definition of the error codes used in this specification.

The API defines the following other constant:

• **Constant Name**	• **Value**
S_MONOTONIC_COUNTER_GLOBAL	0x00000000

B6.2.3 Functions

B6.2.3.1 SMonotonicCounterInit

```
S_RESULT SMonotonicCounterInit(void)
```

Initializes the API. This function is not thread safe. Undefined behavior may occur if multiple threads call it at same time.

Return Value

- `S_SUCCESS` indicates that the monotonic counter API has been successfully initialized
- `S_ERROR_COMMUNICATION` if the secure world is not responding or not accessible
- `S_ERROR_OUT_OF_MEMORY` if no more memory is available to perform the operation

B6.2.3.2 SMonotonicCounterTerminate

```
void SMonotonicCounterTerminate(void)
```

Closes access to the monotonic counter API. This function is not thread safe.

This function waits until all pending functions of the API have completed. After the function has returned, all handles returned by the API are invalid.

B6.2.3.3 SMonotonicCounterOpen

```
S_RESULT SMonotonicCounterOpen(
                 uint32_t nCounterIdentifier,
                 OUT S_HANDLE* phCounter)
```

Opens a connection to a specific monotonic counter.

Currently, only one global shared counter is available. Its identifier is `S_MONOTONIC_COUNTER_GLOBAL` (`0x00000000`).

This function returns an opaque handle that must be passed to all subsequent functions relative to this counter.

If the current counter has not been accessed yet, it is created and its value is initialized with a random number. The most-significant bit of this value is set to 0 so that a range of at least 2^{63}-1 distinct values is guaranteed to be available for the counter. As a consequence, the counter cannot realistically overflow during its life-time.

Parameters

- `nCounterIdentifier`: the identifier of the counter. Currently, must be set to `S_MONOTONIC_COUNTER_GLOBAL`
- `phCounter`: a pointer filled with an opaque handle denoting the connection to the counter. This is set to `S_HANDLE_NULL` upon error

Return Value

- `S_SUCCESS` indicates that the connection has been successful opened. `*phCounter` then contains an opaque handle that can be used in subsequent functions relative to this counter
- `S_ERROR_COMMUNICATION` if the secure world is not responding or not accessible
- `S_ERROR_OUT_OF_MEMORY` if no more memory is available to perform the operation
- `S_ERROR_ITEM_NOT_FOUND`: if the counter identifier does not exist. In this version of the specification, this happens when `nCounterIdentifier` is different from `S_MONOTONIC_COUNTER_GLOBAL`.
- `S_ERROR_STORAGE_UNREACHABLE`: the underlying storage is not accessible. This is likely to be a temporary error. A later attempt may succeed. This may require a reboot of the device however. You may find more information in the *Product Reference Manual* shipped with your product.
- `S_ERROR_STORAGE_CORRUPTED`: a corruption has been detected, either due to a physical failure or an attack. This error is likely to be definitive. A reinitialization of the platform may be necessary. You may find more information in the *Product Reference Manual* shipped with your product
- `S_ERROR_STORAGE_NO_SPACE`: there is not sufficient space to create the counter.

B6.2.3.4 SMonotonicCounterClose

```
void SMonotonicCounterClose(S_HANDLE hCounter)
```

Closes a connection to a monotonic counter. After this function returns, the handle becomes invalid and must not be used anymore.

If `hCounter` is `S_HANDLE_NULL`, this function does nothing.

Parameters

- `hCounter`: a handle on a monotonic counter connection or `S_HANDLE_NULL`. If different from `S_HANDLE_NULL`, the handle must have been returned by `SMonotonicCounterOpen` and must not have been closed yet.

B6.2.3.5 SMonotonicCounterGet

```
S_RESULT SMonotonicCounterGet(
                 S_HANDLE hCounter,
                 S_MONOTONIC_COUNTER_VALUE* psCurrentValue)
```

Retrieves the current value of a counter.

Note that this function may not fail due to out of memory but may detect corruption of the counter.

Parameters

- `hCounter`: a handle on a monotonic counter connection. This handle must have been returned by `SMonotonicCounterOpen` and must not have been closed yet.
- `psCurrentValue`: a pointer filled with the current value of the counter. Upon error, the value `0xFFFFFFFFFFFFFFFF` is returned, i.e., `nLow` and `nHigh` are both set to `0xFFFFFFFF`.

Return Value

- `S_SUCCESS` indicates that the operation succeeded. `*psCurrentValue` is then filled with the current value of the counter
- `S_ERROR_COMMUNICATION` if the secure world is not responding or not accessible
- `S_ERROR_STORAGE_UNREACHABLE`: the underlying storage is not accessible. This is likely to be a temporary error. A later attempt may succeed. This may require a reboot of the device however. You may find more information in the *Product Reference Manual* shipped with your product.
- `S_ERROR_STORAGE_CORRUPTED`: a corruption has been detected, either due to a physical failure or an attack. This error is likely to be definitive. A reinitialization of the platform may be necessary. You may find more information in the *Product Reference Manual* shipped with your product.

B6.2.3.6 SMONOTONICCOUNTERINCREMENT

```
S_RESULT SMonotonicCounterIncrement(
                 S_HANDLE hCounter,
                 S_MONOTONIC_COUNTER_VALUE* psNewValue)
```

Increment the monotonic counter by one and returns the new current value of the counter.

This operation is atomic: either the entire operation completes successfully or it is not performed at all. In case of error, the value of the counter is unchanged.

Note that this function may not fail due to out of memory but may detect corruption of the counter.

Parameters

- `hCounter`: a handle on a monotonic counter connection. This handle must have been returned by `SMonotonicCounterOpen` and must not have been closed yet.
- `psNewValue`: a pointer filled with the new current value of the counter. Upon error, the value `0xFFFFFFFFFFFFFFFF` is returned, i.e., `nLow` and `nHigh` are both set to `0xFFFFFFFF`.

Return Value

- `S_SUCCESS` indicates that the operation succeeded. The counter has been incremented by one and `*psNewValue` has been filled with the new value of the counter
- `S_ERROR_COMMUNICATION` if the secure world is not responding or not accessible
- `S_ERROR_STORAGE_UNREACHABLE`: the underlying storage is not accessible. This is likely to be a temporary error. A later attempt may succeed. This may require a reboot of the device however. You may find more information in the *Product Reference Manual* shipped with your product.
- `S_ERROR_STORAGE_CORRUPTED`: a corruption has been detected, either due to a physical failure or an attack. This error is likely to be definitive. A reinitialization of the platform may be necessary. You may find more information in the *Product Reference Manual* shipped with your product.

Chapter B7 SERVICE MANAGER PROTOCOL

This chapter is the specification of the Service Manager protocol. This protocol provides access to the Service Manager built-in service from Normal-World applications using the TF Client API and from Secure-World services using the `SXControl` API.

The Service Manager allows enumerating the installed services and their properties and, in some implementations, also allows downloading or removing services.

The Service Manager may or may not be available, or may have restricted functionality depending on the configuration of the Secure World software:

- If the Secure World software has a fixed configuration, then there may be no Service Manager interface.

If the Secure World software has no capability to dynamically download or remove services, then manager-level access (see below) may be unavailable, and attempts by clients to connect to the Service Manager with this level of access may be rejected.

Protocol Overview

The Service Manager service supports the following commands:

Name	Value	Description
Open	n/a	Opens a connection with the Service Manager
`SERVICE_MANAGER_COMMAND_ID_GET_ALL_SERVICES`	`0x00000000`	Gets a the list of all the available services UUIDs
`SERVICE_MANAGER_COMMAND_ID_GET_PROPERTY`	`0x00000003`	Gets a service property
`SERVICE_MANAGER_COMMAND_ID_GET_ALL_PROPERTIES`	`0x00000004`	Gets all the properties of a service
`SERVICE_MANAGER_COMMAND_ID_DOWNLOAD_SERVICE`	`0x00000001`	Downloads a new service

SERVICE_MANAGER_COMMAND_ID_REMOVE_SERVICE	0x00000002	Removes a service
Close	n/a	Closes the connection to the service manager

When a command identifier not listed above is received by the Service Manager, it fails with the error code S_ERROR_NOT_SUPPORTED.

B7.1 OPEN OPERATION

Service UUID

`61419A20-33FE-42AB-85Fe-88B0EEFBA004`

Login Type

The login types are treated as follows:

- `S_LOGIN_PRIVILEGED` allows the client to get user-level or manager-level access;
- `S_LOGIN_AUTHENTICATION` allows the client to get user-level and manager-level access if the client's manifest (see section B3.3.6.2) contains the property "`sm.service_manager.allow_manager_mode`" with the value set to "`true`";
- all other login types: grant user-level access only.

Parameters

- Param#0: `VALUE_INPUT`:
    - o `a`: `nAccessMode`

      The Service Manager access mode, which determines the level of access to the Service Manager (See below.)
    - o `b`: ignored
- Param #1 and #2: `NONE`
- Param #3: `NONE` unless the login type `S_LOGIN_AUTHENTICATION` is used, in which case it is of type `MEMREF_INPUT` and contains the signature file

`nAccessMode` must have one of the following values:

- `S_CONTROL_MODE_USER (0x00000002)`:

  Requests user-level access. A Service Manager session opened with user-level access permits list of installed services and the properties of services to be queried, but does not permit the download or removal of services.
- `S_CONTROL_MODE_MANAGER` (`0x00000008`):

  Requests manager-level access. A session opened with manager-level access can perform any operation supported by the Service Manager.
- `S_CONTROL_MODE_EXCLUSIVE_MANAGER` (`0x00000010`):

  Requests manager-level access, and requires that no other client can be connected with manager-level access for the lifetime of the session.

Note that the manager or exclusive-manager modes are available only for products that support the dynamic downloading and removal of secure services.

Effect

The error code can be one of the following:

- `S_SUCCESS`

  The Service Manager session was opened successfully, with the requested access type.
- `S_ERROR_OUT_OF_MEMORY`

  Not enough resource is available to open the session
- `S_ERROR_BAD_PARAMETERS`

  The parameter types are incorrect.
- `S_ERROR_ACCESS_DENIED`

  The Service Manager is not present, the login was rejected, or the requested access type was not granted because one of the following occurred:

  - manager-level access was requested but is not supported by the Service Manager because the product does not support dynamic downloading and removal of secure services;
  - manager-level access was requested but could not be granted because the login type did not permit it (see above).
- `S_ERROR_EXCLUSIVE_ACCESS`

  Manager-level access was requested and the login type permitted it, but either exclusive manager access was requested and some client was already connected with manager-level access, or some client was already connected with exclusive manager-level access.

B7.2 'Get All Services' Command

Requests the list of installed services to be returned to the client.

Command ID

- `SERVICE_MANAGER_COMMAND_ID_GET_ALL_SERVICES`

Parameters

- Param #0: must be of type `MEMREF_OUTPUT`
- Param #1-#3: must be of type `NONE`

Effect

The Service Manager copies an array with the UUIDs of all the services in the parameter #0.

If the parameter is `NULL` or if its size is not large enough to contain all the UUIDs, then no data is written in the parmater #0 but its size is updated with the required number of bytes to store all the UUIDs and the error `S_ERROR_SHORT_BUFFER` is returned.

Otherwise, the size of parameter #0 is updated with the actual number of bytes stored in the buffer and `S_SUCCESS` is returned.

Error Codes

- `S_SUCCESS` if the operation was successful
- `S_ERROR_SHORT_BUFFER` if the buffer in parameter #0 is not large enough to hold all the service UUIDs
- `S_ERROR_BAD_PARAMETERS` if the parameters are not exactly of the types specified above

B7.3 'GET SERVICE PROPERTY' COMMAND

Queries a specific named property of an installed service.

Note that only the "public" properties of a service are visible through the Service Manager. See section B2.3.6.1, "*Service Properties*" for more details.

Command ID

- `SERVICE_MANAGER_COMMAND_ID_GET_PROPERTY`

Parameters

- Parameter #0 must be of type `MEMREF_INPUT` and must contain exactly 16 bytes. It contains the UUID of the service which is being queried
- Parameter #1 must be of type `MEMREF_INPUT` and must not be `NULL`. It contains the name of the requested property
- Parameter #2 must be of type `MEMREF_OUPUT`. It points to a buffer to get the value of the requested property. It can be `NULL` to get the size of this property.
- Parameter #3: not used, must be of type `NONE`.

Effect

The service is found by its UUID then the requested property is looked up. The following error codes can be returned:

- `S_SUCCESS`

  The property was queried successfully. The size of parameter #2 is updated with the actual number of bytes in the value of the property. Note that this does not include any zero terminator.
- `S_ERROR_SHORT_BUFFER`

  The service and the property was found but the parameter #2 is not large enough to contain the value of the property. In this case, no data is written in parameter #2, but its size is updated to reflect the actual number of bytes necessary to hold the value of the property.
- `S_ERROR_ACCESS_DENIED`

  No service with the specified UUID is installed.
- `S_ERROR_ITEM_NOT_FOUND`

  A matching service is installed, but no property with a name matching Parameter #1 was found.
- `S_ERROR_BAD_PARAMETERS`

  The type and size of the parameters is not exactly as specified above.

B7.4 'GET ALL SERVICE PROPERTIES' COMMAND

Requests that all the properties of an installed service be returned to the client.

Note that only the "public" properties of a service are visible through the Service Manager. See section B2.3.6.1, "*Service Properties*" for more details.

Command ID

- `SERVICE_MANAGER_COMMAND_ID_GET_ALL_PROPERTIES`

Parameters

- Parameter #0 must be of type `MEMREF_INPUT` and contains exactly 16 bytes. It contains the UUID of the service to query.
- Parameter #1 must be of type `MEMREF_OUTPUT`. It is an output buffer filled with the service properties in a format described below. The buffer must be aligned on a four-byte boundary.
- Parameter #2-#3 are not used and must be of type `NONE`.

Effect

The service with the required UUID is looked up, then all the properties of this service, in no specific order, are written in the output buffer:

- The format of one property is as follows:
  - `uint32_t nPropertyNameLength`

    The number of bytes in the property name. This does not include any zero terminator.
  - `uint32_t nPropertyValueLength`

    The number of bytes in the property value. This does not include any zero terminator.
  - `uint8_t sPropertyName[nPropertyNameLength]`

    The property name.
  - `uint8_t sPropertyValue[nPropertyValueLength]`

    The property value
- The format of the whole output buffer is the concatenation of all the properties, with 0-3 bytes of padding written between each property to ensure that the start of the new property (field `nPropertyNameLength`) is aligned on a four-byte boundary.

If the output buffer is not large enough to contain all these data, then nothing is written in the parameter #1, but its size is updated to reflect the number of bytes necessary to contain all the properties and the error code `S_ERROR_SHORT_BUFFER` is returned.

The following error codes may be returned by this command:

- `S_SUCCESS`

  The service was found and all the properties were copied successfully in the parameter #1.
- `S_ERROR_SHORT_BUFFER`

  The service was found, but the parameter #1 is not large enough to hold all the properties. Its size has been updated with the necessary number of bytes.
- `S_ERROR_BAD_PARAMETERS`

  The parameters types, sizes, or alignment are not exactly as specified above.
- `S_ERROR_ACCESS_DENIED`

  No service with the specified UUID is installed

B7.5 'DOWNLOAD SERVICE' COMMAND

Requests the download of a new service into the Secure World.

Manager-level access is required in order to perform this operation. This means that:

- Manager access must have been requested when the session was opened
- Manager access must have been granted to the client by the Service Manager
- This in particular means that the implementation must support dynamic downloading and removal of secure services

Command ID

- `SERVICE_MANAGER_COMMAND_ID_DOWNLOAD_SERVICE`

Parameters

- Parameter #0: must be of type `MEMREF_INPUT`, contains the download file `pDownloadFile`
- Parameter #1: must be of type `MEMREF_OUTPUT` and contain exactly 16 bytes. This parameter is filled with the downloaded service UUID
- Parameters #2-#3: unused, must be of type `NONE`

Effect

The download file specified in parameter #0 is copied into secure memory and some implementation-dependent processing is performed to check its signature and its content and to install the new service.

The following error codes can be returned:

- `S_SUCCESS`

  The new service was downloaded and installed successfully. In this case, the parameter #1 is filled with the UUID of the new service.
- `S_ERROR_BAD_PARAMETERS`

  The parameter types and sizes are not exactly as specified above or if the service was correctly signed but otherwise ill-formed.
- `S_ERROR_OUT_OF_MEMORY`

  Not enough resource is available to install the new service.
- `S_ERROR_ACCESS_DENIED`

  The service could not be installed because:
  - the client is not connected with manager-level access;
  - the service is not correctly signed; or
  - a service with a UUID matching that of the new service is already installed.

B7.6 'Remove Service' Command

Requests the removal of a service from the Secure World.

Manager-level access is required in order to perform this operation. This means that:

- Manager access must have been requested when the session was opened
- Manager access must have been granted to the client by the Service Manager
- This in particular means that the implementation must support dynamic downloading and removal of secure services

Command ID

- `SERVICE_MANAGER_COMMAND_ID_REMOVE_SERVICE`

Parameters

- Parameter #0 must be of type `MEMREF_INPUT` and contain exactly 16 bytes. It contains the UUID of the service to be removed
- Parameters #1-#3 are not used and must be of type `NONE`

Effect

The service with the specified UUID is removed. The following error codes can be returned:

- `S_SUCCESS`

  The service was removed successfully.
- `S_ERROR_BAD_PARAMETERS`

  The parameter types or sizes are not exactly as specified above.
- `S_ERROR_ITEM_NOT_FOUND`

  The service could not be removed because no service with the specified UUID is installed
- `S_ERROR_ACCESS_DENIED`

  The service could not be removed because:
  - the client is not connected with manager-level access or
  - the service cannot be removed at the present time (because it is a built-in service or is currently in use).

Appendix I. VERSION HISTORY

This is the list of V3.x APIs issued so far.

Refer to your *Product Reference Manual* to determine which API version is implemented. You also can use the implementation property "`ssdi.apiversion`" to dynamically determine the API version.

I.1 VERSION 3.1

The Cryptographic API in V3.1 introduces the possibility to create or generate a non-sensitive RSA private key. When a RSA private key is non-sensitive, it is possible to retrieve its protected attributes, for example the private exponent. See section B4.6.1.2, "*RSA private key objects*" for more details.

APIs V3.0 and V3.1 are compatible in the following sense:

- All code developed for the APIs V3.0 works unchanged on a product that implements the APIs V3.1. It is also possible to upgrade a device to a V3.1 product without affecting the existing RSA private key objects
- All code developed for the APIs V3.1 works unchanged on a V3.0 product if it does not attempt to set the `CKA_SENSITIVE` attribute for RSA private key objects. If it does set the `CKA_SENSITIVE` attribute, even to `CK_TRUE`, the calls to `C_CreateObject` or `C_GenerateKeyPair` fail with the error code `CKR_ATTRIBUTE_TYPE_INVALID`.

I.2 VERSION 3.0

Version 3.0 is a major revision of the Trusted Foundations Developer APIs.

For more information on how to migrate code developed against the V2.x APIs to a product implementing the V3.x APIs, see the *Developer Compatibility Guide* **[CompatGuide]**.

Appendix II. REFERENCES

Add GlobalPlatform TEE Client API spec.

[GPTEEC]	Global Platform Device Technology, "*TEE Client API Specification*". GPD_SPE_007. Public Release, July 2010.
[CompatGuide]	Trusted Logic, "*Developer Compatibility Guide*", CP-2010-RT-683.
[FIPS 46-3]	FIPS PUB 46-3: Data Encryption Standard (DES) http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
[FIPS 180-3]	NIST. FIPS PUB 180-3: "*Secure Hash Standard (SHS)*" October 2008 http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
[FIPS 197]	FIPS PUB 197: Advanced Encryption Standard (AES) http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[FIPS 198]	FIPS PUB 198: The Keyed-Hash Message Authentication Code (HMAC) http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf
[NIST800-38A]	Recommendation for Block Cipher Modes of Operation NIST Special Publication 800-38A – 2001 Edition
[PKCS#1]	RSA Security Inc, *"PKCS#1: RSA Encryption Standard"*, v2.1, http://www.rsa.com/rsalabs/node.asp?id=2125
[PKCS#3]	RSA Security Inc, PKCS#3: Diffie-Hellman Key Agreement Standard ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-3.asc
[PKCS#7]	RSA Security Inc, *"PKCS#7: Cryptographic Message Syntax Standard"*, v1.5, http://www.rsa.com/rsalabs/node.asp?id=2129
[PKCS#8]	RSA Security Inc, *"PKCS#8: Private-Key Information Syntax Standard"*, v1.2, http://www.rsa.com/rsalabs/node.asp?id=2130
[PKCS#11]	RSA Security Inc, PKCS#11: Cryptographic Token Interface Standard, version 2.20 http://www.rsasecurity.com/rsalabs/node.asp?id=2133
[RFC1321]	R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992

	http://ietf.org/rfc/rfc1321.txt
[RFC3280]	R. Housley, W. Polk, W. Ford, D. Solo, *"Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile"*, http://www.ietf.org/rfc/rfc3280.txt
[RFC3629]	RFC 3629, "UTF-8, a transformation format of ISO 10646", Internet Engineering Task Force http://www.ietf.org/rfc/rfc3629.txt
[RFC3686]	Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP) http://www.ietf.org/rfc/rfc3686.txt
[RFC4122]	P. Leach, M. Mealling, R. Salz, *"A Universally Unique IDentifier (UUID) URN Namespace"*, http://www.ietf.org/rfc/rfc4122txt