Qualcomm Technologies, Inc.

QTEE 5.0 TrustZone Development Application Note

80-NM090-2 C

June 12, 2018

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	January 2018	Initial release
B	April 2018	Added note to Section 2.1.4.1
C	June 2018	Updated QTEE version 5.0 and added SM8150 chipset to Section 1.1

Contents

Tables

1 Introduction

1.1 Purpose

This document describes development security guidelines for the Qualcomm® Trusted Execution Environment (QTEE) ARM TrustZone (TZ) application.

This document is written for engineers designing and developing QTEE applications. It complements the API information found in the *QTEE Version 5.0 User Guide (80-NH537-4)* by providing guidelines and best practices for correctly using these APIs in QTEE applications. This document is applicable to the SDM845 and SM8150 chipsets that support TZ application development.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 QTEE run-time environment

2.1 QTEE application feature updates

Most of the improvements made to the QTEE 5.0 run-time environment are intended to improve the future binary compatibility. Due to these improvements, the applications built for earlier releases that are not binary compatible with QTEE 5.0 and are required to be rebuilt and updated in some cases. The following sections highlight changes to be aware of when adapting or porting applications to QTEE 5.0.

2.1.1 IDL compiler

A significant improvement to the QTEE TA build environment is the integration of the `minkidl` Mink IDL compiler, into the build system. With the introduction of the `minkidl` compiler, IPC interfaces are now written in Mink IDL (`.idl` files), which when compiled autogenerates the IPC source files. This simplifies writing TA to TA interfaces by providing a structured syntax that uses a simple set of types with the goal of improved security and compatibility. The build environment is updated to transparently compile IDL sources and install the resulting IPC headers.

2.1.2 Shared dynamic commonlib library

Starting with QTEE 5.0, the commonlib TA interface library is now a standard ELF dynamic shared library that provides shared interfaces for all TAs. Depending on the architecture of the TA, it automatically links against either the 32-bit `libcmnlib.so` or the 64-bit `libcmnlib64.so` shared library when using `SecureAppBuilder`.

2.1.2.1 Source and binary compatibility

As part of the library update, the availability of exported TA interfaces and symbols is reduced to only those that are documented and are available in the public QTI-provided API header files. QTI only provides backwards compatibility assurance for the documented symbols. Inclusion of any header files or use of interfaces or symbols outside this published set are considered unsupported and can potentially break future compatibility. To avoid future compatibility issues, TAs should limit inclusion of header files to the following default QTEE public headers paths:

```
ssg/api/securemsm/
ssg/securemsm/trustzone/qsee/include/
```

In some case, previously available interfaces are deprecated and are no longer available. See Section 2.3.3 for a list of these interfaces.

2.1.2.2 Library binary images

The libraries are located as follows:

```
> cd trustzone_images/
> ls ssg/bsp/trustzone/qsapps/commonlib/build/WAXAANAA/*.so
ssg/bsp/trustzone/qsapps/commonlib/build/WAXAANAA/libcmnlib.so
> ls ssg/bsp/trustzone/qsapps/cmnlib64/build/WAXAANAA/*.so
ssg/bsp/trustzone/qsapps/cmnlib64/build/WAXAANAA/libcmnlib64.so
```

The commonlib binary image filenames maintain their previous names, therefore no nonsecure client-side changes are required to load the commonlib images. The binary images are still located as follows:

```
> cd trustzone_images/
> ls build/ms/bin/WAXAANAA/*cmnlib*.mbn
build/ms/bin/WAXAANAA/cmnlib64.mbn
build/ms/bin/WAXAANAA/cmnlib.mbn
```

2.1.3 Trusted application identification

QTEE 5.0 has introduced the concept of a distinguished name and IDs to ensure unique TA identity across multiple signing authorities.

2.1.3.1 Distinguished name

A TA distinguished name is a dot-delimited textual identifier that is generated by combining components of the TA's certificate chain along with certain application metadata. The combination allows for the creation of a unique name where modifying any component results in a different name and thus is considered a different application from a security standpoint. The number of fields and field types of a distinguished name are dependent on the TA signing authority as well as the availability and uniqueness of certain TA metadata.

OEM-signed TAs

The distinguished name of an OEM-signed TA depends on whether the TA has defined a UUID or not as part of its metadata. If present then the UUID takes priority over the application name when generating the distinguished name and has the following format:

```
oem.<UUID>
```

For example, an OEM-signed application with a UUID of "b9a20333-2126-4f41-872b-fcda3454b896" has the following distinguished name:

```
oem.b9a20333-2126-4f41-872b-fcda3454b896
```

If the OEM-signed TA does not define a UUID, then the TA name as defined in the application metadata is used instead and expected to have the following format:

```
oem.<appName>
```

An example of a distinguished name given to an OEM-signed application with the name "oemappname" is as follows:

```
oem.oemappname
```

NOTE: For legacy reasons, the qualified naming rules are voided and the certificate chain field is removed for any application signed by a trusted root certificate, although it is still supported in lookup scenarios. Only OEM certificates are considered trusted at the present.

Alternate root signed TAs

In the case of TAs signed with an alternate-root, the distinguished name follows the same naming conventions of an OEM-signed TA, but adds the hash of the root certificate as an extra component to guarantee uniqueness. This is similar to the information that is added to the oem_secapp.xml file when specifying alternate signing roots. The added component results in one of the following distinguished name formats for TAs signed with an alternate-root:

```
alt.<alt cert hash>.<UUID>
alt.<alt cert hash>.<appName>
```

Examples of distinguished names given to alternate-signed TAs are as follows:

```
alt.0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20.
491dfeab-df8b-47f4-911f-8bf06e725e87

alt.0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20.alt
appname
```

One is defined with a UUID of “491dfeab-df8b-47f4-911f-8bf06e725e87“and the other with an application name of “altappname”.

2.1.3.2 Distinguished ID

In addition to the TA distinguished name, the concept of a distinguished ID is also new to QTEE 5.0. A distinguished ID is a binary representation of the distinguished name and is primarily used for key generation, access control and as a general application identifier.

2.1.3.3 Usage and backward compatibility

Throughout the QTEE 5.0 environment, distinguished names and IDs have replaced the use of other identifiers. The following features and functionality are improved to use the new identifiers:

- The rollback protection version table data now uses the distinguished ID in place of the certificate application ID.
- OEM configuration XML data is now required to use the distinguished name format, as defined in the preceding sections, for identifying a TA.
- HLOS TA lookup is now performed using a name argument that follows the distinguished naming conventions.
- Identifying the source TA of a message retrieved from the QTEE bulletin board service is now performed using distinguished naming.
- Identifying the source TA of an interapplication message is now performed using distinguished naming.

- TA cryptographic keys are now derived from the application distinguished ID and as a result are no longer compatible with keys from releases before QTEE 5.0.
- Secure storage is now encrypted using the new TA cryptographic keys and as a result is incompatible with keys from releases before QTEE 5.0.

2.1.4 Android SFS and GP persistent object changes

The following improvements are made to the secure file system (SFS) and the global platform (GP) persistent object subsystems in QTEE 5.0 for Android. These changes should be considered when updating a TA for QTEE 5.0.

2.1.4.1 Trusted application SFS file persistence

Per-TA SFS file persistence is added to allow finer granularity control over whether file system data gets destroyed on a factory reset. By default, TA SFS file system data is preserved across a factory reset. This behavior can now be modified on a per-TA basis by setting the new TA `storageFilesNoPersist` metadata property. See Section 2.2.1 for more details on this metadata property.

A side-effect of using the storage `storageFilesNoPersist` metadata property is that it affects the root path of the TA SFS files. The difference in the root path ultimately decides whether the content is preserved or not. Table 2-1 outlines the difference in the root path assignment depending on the `storageFilesNoPerisist` property setting. The base TA folder is added to the root path as described in Section 2.1.4.3 .

NOTE: The root paths can vary from one HLOS to another as they are determined by the file system listener implementation. Also, in some cases a partition to store data persistently may not be available. Some HLOS like Android currently allows OEMs to customize the root paths.

The following root paths are the default for Android and can be modified by the OEMs.

Table 2-1 SFS file persistence path

storageFileNoPersist	Root path
True	/data/vendor/tzstorage
False (default)	/persist/data

Due to the incompatibility with this feature, the OEM control of the SFS root path via the `cmnlib_oem_config.xml` file is deprecated. Specifically, the `gppo_root_path` and `cmnlib_gppo_root_path` properties are no longer supported and are ignored if used.

2.1.4.2 Increased file limit

The maximum number of SFS files supported for a given TA has increased from the QSEE 4.0 limit of 56 files up to a TA defined maximum. By using the new `totalStorageFiles` metadata property, a TA can increase the file limit based on its expected usage. Not setting the property results in a default maximum file limit of 60

files. See Section 2.2.1 details on setting this metadata property. This feature is verified with a limit of up to 2000 files.

2.1.4.3 Encoded file and directory names

SFS is updated and no longer uses the TA name when naming the SFS file directory and index files. Instead, the TA directory and index file naming have improved to be more unique and is obfuscated by instead using an encoded name that looks more like the following:

Example: Storing files in nonpersistent partition and using the default root path.

```
> cd /data/vendor/tzstorage
> ls
k6SwcltMUbt7Oc0LSIGsXV6ho+56UmmvKoFpO27BfEE_

> cd k6SwcltMUbt7Oc0LSIGsXV6ho+56UmmvKoFpO27BfEE_/
> ls
eKEgHx3jo+L6bl9F4vQb
eKEgHx3jo+L6bl9F4vQb.bak
k6SwcltMUbt7Oc0LSIGsXV6ho+56UmmvKoFpO27BfEE_
k6SwcltMUbt7Oc0LSIGsXV6ho+56UmmvKoFpO27BfEE_.bak
```

2.1.5 Shared memory changes

2.1.5.1 Shared memory privileges

Whitelist memory validation of shared memory was introduced before QTEE 5.0 to address shared memory security concerns. The `RTIC` privilege was added at the same time to facilitate backward compatibility of TAs affected by whitelist validations and to enable shared memory use cases where the shared memory regions are not passed by the client application.

With QTEE 5.0, the `RTIC` privilege is renamed to `WhitelistBypass` and retains the same functionality. In addition, a new privilege, `WhitelistBypassRO` is also introduced to restrict TAs to read-only access for memory not within a whitelist. See Section 2.2.2 for more details on both the privileges.

2.1.5.2 Shared memory interfaces

`IMemSpace` is the new TA interface used to share memory with client applications. The interface requires shared memory regions to be described by using Mink memory objects. In the absence of support for passing shared memory objects from non-secure client applications using mink IPC, TAs instead create memory objects using the newly added `IHlosRegionFinder` interface by passing it physical memory addresses.

The `IHlosRegionFinder` interface performs memory validations and if successful returns a memory object that represents the capability to access a shared memory region. Access to a shared memory region is either read-only or read-write depending on validation results and TA privileges.

- Default (no whitelist privilege): Whitelist validation is performed and write access is determined by nonsecure stage-2 translation table memory permissions.
- `WhiteListBypassRO`: Whitelist validation is performed. If successful, write access is determined by non-secure stage-2 translation table memory permissions. If unsuccessful, the returned region is read-only.
- `WhitelistBypass`: Whitelist validation is skipped and a read-write region is returned.

Table 2-2 TA whitelisting privilege vs. resulting permission

TA privilege set	Whitelisted		Unwhitelisted	
	HLOS S2 RW	HLOS S2 RO	HLOS S2 RW	HLOS S2 RO
Default	RW	RO	None	None
WhitelistBypassRO	RW	RO	RO	RO
WhitelistBypass	RW	RW	RW	RW

Memory region objects returned by `IHlosRegionFinder` are generic mink objects. They may be invoked using the `IMemRegion` interface or passed to other domains that accept `IMemRegion` objects as part of a mink IPC interface.

Unlike `qsee_register_shared_buffer`, the `IHlosRegionFinder` interface provides no guarantee of exclusivity to a shared memory region. When a TA establishes a shared memory region using either `qsee_register_shared_buffer` or `IHlosRegionFinder`, other TAs may use `IHlosRegionFinder` to access the same shared memory region.region If a TA has gained access to a shared memory region through `qsee_register_shared_buffer`, other TAs will not be able to access the same shared memory region using `qsee_register_shared_buffer`.

The pool of available memory region mappings is shared between the `IHlosRegionFinder` and `qsee_register_shared_buffer` interfaces. As a result, the interfaces also share the limit on the number of memory regions that are simultaneously mapped. This mapping limit is not enforced on memory region object creation but rather when attempting to map a created region into the TA's memory space, as described in Section 2.4.2.1 Memory region object creation, as described in Section 2.4.2 is however limited by wider system memory constraints.

2.1.6 Dynamically allocated TA memory

In QSEE 4.0, the TA stack and heap ranges were statically allocated as part of the TA binary layout that was generated based on TA builder parameters. Equivalently, the TA accept buffer was also declared statically based on the TA metadata. To improve upon the security of these memory buffers, these resources are now dynamically allocated during TA loading based on new TA metadata size properties rather than as previously declared via build parameters. This improvement allows for better memory containment as well as randomized placement.

The two new TA metadata properties added in QTEE 5.0 are `heapSize` and `stackSize`, and their use has replaced the use of the similarly named TA builder parameters `heap_size` and `stack_size`, respectively. See Section 2.2.1 for details pertaining to the new metadata properties. Given that the TA accept buffer was already sized using the

taAcceptBufSize metadata property, no new property was needed, or any configuration changes required for it. The following example demonstrates the replacement of the SecureAppBuilder parameters with the equivalent metadata declarations.

```
app_units = env.SecureAppBuilder(
  sources = sources,
  includes = includes,
  metadata = md,
  image = target_name,
  heap_size = '0x10000'          ← Remove old heap size declaration
  stack_size = '0x20000',        ← Remove old stack size declaration

user_libs = libs
)
```

Must be updated to instead declare the sizes in their metadata:

```
md = {
  'appName':    app_name,
  'privileges': ['default'],
  'heapSize':  0x10000,          ← Add new heap size declaration
  'stackSize': 0x20000,          ← Add new stack size declaration
  'acceptBufSize': 0x4000        ← Declared the same as previously
}
```

As was the case with the past declarations, these parameters are optional and only necessary if a heap or stack size greater than the default values is required. Use of the SecureAppBuilder parameters is now deprecated and continued use results in a build failure such as:

```
Error: Declaring stack size via SecureAppBuilder is no longer supported.
       Please declare stack size in the app metadata.
```

2.1.7 Increased TA heap usage

The demand on the TA heap has increased due to the restructuring of the TA build environment and new feature usage. The default TA heap size is increased in QTEE 5.0 from 32 KB to 56 KB to accommodate the decrease in the amount of statically allocated TA data used. In addition to this increase, some TAs may need to further extend their heap depending on the following features.

NOTE: TAs previously using a custom, nondefault, heap size are advised to increase their heap size by the 24 KB difference mentioned in the previous section to avoid a shortage.

2.1.7.1 SFS heap usage

As described in Section 2.1.4.2, the maximum number of SFS files allowed by a given TA can be increased. To accommodate management of the increased number of files, TAs must also increase their heap size accordingly. The default heap size is already configured to account for management of the default file limit, so TAs only need to adjust

for changes above this limit. The following formula is used to calculate the required amount of increase, in page-size multiples, to the TA heap size for an increase in the SFS file limit over the default:

```
heapSize += ROUNDUP_4K(97 * (totalStorageFiles – 60))
```

For example, if `totalStorageFiles` is increased to 200, then the `heapSize` needs to be increased by 16 KB, which is (97 * (200-60)) = 13.5 KB rounded up to the nearest page multiple. This brings the new TA heap size to 72 KB (56 KB + 16 KB). See Section 2.2.1 for details pertaining to the new `heapSize` and `totalStorageFiles` metadata properties.

2.2 Metadata updates

2.2.1 New metadata properties

Heap size

The `heapSize` metadata property is added for specifying the amount of dynamically allocated heap memory needed by the TA. Within the TA, this property value is obtained by using the `gpd.ta.dataSize` property name with the `TEE_GetProperty*` interfaces. This property has a default of `0xE000` or 56 KB. The property, specified in bytes, can be updated to adjust the amount of heap memory allocated to the TA. The size is automatically rounded up to the nearest page-size multiple during allocation. Set the heap size to 64 KB as follows:

```
'heapSize' : 0x10000
```

Stack size

The `stackSize` metadata property is added to specify the amount of dynamically allocated stack memory needed by the TA. Within the TA, this property value can be obtained by using the `gpd.ta.stackSize` property name with the `TEE_GetProperty*` interfaces. This property has a default of `0x8000` or 32 KB. The property, specified in bytes, is updated to adjust the amount of heap memory allocated to the TA. The size is automatically rounded up to the nearest page-size multiple during allocation. Set the heap size to 64 KB as follows:

```
'stackSize' : 0x10000
```

Cryptography algorithm self-test

The `cryptoSelfTest` metadata property is added to enable the self-test for FIPS certifiable crypto algorithms if needed by the TA. If enabled, self-tests are performed during TA loading and if not passed results in the failure to load. This property has a default of `False`. Set the property to `True`, to enable self-test of the crypto algorithms before loading the TA:

```
'cryptoSelfTest' : True
```

SFS file persistence

The `storageFilesNoPersist` metadata property is added to allow a TA to specify whether its SFS files should persist across a factory reset. The default of this property is

`False`. Set the property to `True`, to have the TA's SFS files getting deleted in the event of a factory reset. The setting of this property also affects the TA SFS file root path as described in Section 2.1.4.1.

```
'storageFilesNoPersist' : True
```

SFS file limit

The `totalStorageFiles` metadata property is added to allow a TA to specify its maximum SFS file count. By default, the maximum file count is 60. Set the property to a different unsigned integer value to modify the enforced file limit. For example, setting the property as follows increases the SFS file limit to 2000 files.

```
'totalStorageFiles' : 2000
```

Version

The `version` metadata property is added to allow a TA to specify its version number in string format. Within the TA, this property value can be obtained by using the `gpd.ta.version` property name with the `TEE_GetProperty*` interfaces. Set the property as follows to specify the TA version is 1.0.

```
'version' : '1.0'
```

Description

The `description` metadata property is added to allow a TA to provide a string describing the TA. Within the TA, this property value can be obtained by using the `gpd.ta.description` property name with the `TEE_GetProperty*` interfaces. Set the property as follow to specify the description of the app as My App.

```
'description' : 'My App'
```

Custom properties

The custom `properties` metadata property is added to declare custom TA properties. Within the TA, the custom property value can be obtained by using its name with the `TEE_GetProperty*` interfaces. Set the custom properties as follows to specify the two TA custom properties "foo" and "bar".

```
'properties' : { 'foo': 'this is a test string',
                 'bar': 'true'}
```

GP multisession support

The `multiSession` metadata property is added to indicate whether a GP TA supports multiple sessions or not. Within the TA, the property value is obtained via the GP property interface using the `gpd.ta.multiSession` property. The default of this property is `False`. Setting the property value, as follows, to `True` enables GP multi-session support.

```
'multiSession' : True
```

2.2.2 New metadata privileges

HlosRegionFinder privilege

The `HlosRegionFinder` privilege is part of the default privilege set granted to all TAs that allows them to use the `IHlosRegionFinder` interfaces for acquiring HLOS memory regions.

WhitelistBypass privilege

The `WhitelistBypass` privilege is used to grant a TA permission to bypass whitelist memory validation checks. This validation is bypassed during shared memory registration when using the legacy `qsee_register_shared_buffer` interface or when acquiring memory regions using the new `IHlosRegionFinder` interface. Due to the potential risk of the privilege, this privilege is not part of the default privilege set granted to all TAs.

This privilege is added as a replacement to the RTIC privilege introduced before QTEE 5.0. This privilege is functionally compatible to its predecessor, but has the added advantage of working with the new `IHlosRegionFinder` interface. See Section 2.2.3 for details pertaining to the replacement of the `RTIC` privilege and Section 2.1.5.2 for details of how whitelist bypass privileges control `IHlosRegionFinder` memory validations and permissions.

NOTE: This privilege should only be granted where necessary as it creates extra risk by circumventing validation checks.

WhitelistBypassRO privilege

The `WhitelistBypassRO` privilege is used to grant a TA reduced permission to access a memory region acquired using the `IHlosRegionFinder` interface despite failing whitelist validation on the region. This privilege has no effect on whitelist address validation on calls to the legacy `qsee_register_shared_buffer` interface. Due to the potential risk of the privilege, this privilege is not part of the default privilege set granted to all TAs.

The `WhitelistBypassRO` privilege is added as a more secure alternative to the `WhitelistBypass` privilege by only allowing read-only permission to any shared memory regions that fail whitelist validation performed by the `IHlosRegionFinder` interface. See Section 2.1.5.2 for details of how whitelist bypass privileges control `IHlosRegionFinder` memory validations. The `WhitelistBypassRO` privilege should be used in preference to the `WhitelistBypass` privilege wherever possible.

NOTE: This privilege should be granted, if necessary as it creates extra risk by circumventing validation checks.

2.2.3 Deprecated metadata privileges

RTIC privilege replaced

The run-time integrity check privilege (`RTIC`) is replaced in QTEE 5.0. Use of the `RTIC` privilege in TA metadata should be updated to either the `WhitelistBypass` or `WhitelistBypassRO` privilege as follows.

```
md = {
   'appName':    app_name,
   'privileges': ['default',
                  'RTIC'],
}
```

Should be updated as follows to get the same behavior:

```
md = {
   'appName':    app_name,
   'privileges': ['default',
                  'WhitelistBypass'],
}
```

Alternatively, the `RTIC` privilege is replaced with the `WhitelistBypassRO` privilege for improved access restrictions. See Section 2.2.2 for more details on the difference and use of these privileges.

2.3 API updates

2.3.1 New APIs

While several new interfaces are added to QTEE 5.0, the following are most notable for features detailed in this document. See Section 2.5, for other interface additions.

qsee_get_space

The `qsee_get_space` API is added to allow a TA to acquire its IMemSpace object. The acquired address space object is used to manage the TA address space.

Syntax:

```
Object qsee_get_space()
```

Parameters:

```
None
```

Returns:

```
IMemSpace object if successful, otherwise NULL
```

2.3.2 Modified APIs

2.3.2.1 Fast Trusted Storage APIs

The fast trusted storage (FTS) interface introduced before QTEE 5.0 are renamed along with their header files to reflect their FTS association. The new interface prototypes are found in the newly added `qsee_fts_*.h` headers that replace the previous `gpPersistObj*.h` private headers where their predecessors resided.

To provide backwards source-compatibility support for the legacy `gpPersistObj*.h` headers and interfaces, temporary headers by the same name is made public and the legacy interfaces are converted to wrapper definitions around the new interfaces. These headers and interfaces are however temporary and it is highly recommended that users switch their usage to the new headers and interfaces as shown in the following tables.

Table 2-3 FTS header update

Interfaces	Old	New
FTS crypto	gpPersistObjCrypto.h	qsee_fts_crypto.h
FTS file I/O	gpPersistObjFileIO.h	qsee_fts_fileio.h

Table 2-4 FTS crpto interface updates

Old	New
gpCrypto_Decrypt	qsee_fts_decrypt
gpCrypto_Encrypt	qsee_fts_encrypt
gpCrypto_Integrity_Protect	qsee_fts_integrity_protect
gpCrypto_Integrity_Verify	qsee_fts_integrity_verify

Table 2-5 FTS file I/O interface updates

Old	New
gpFileIO_Read_File	qsee_fts_read_file
gpFileIO_Remove_File	qsee_fts_remove_file
gpFileIO_Rename_File	qsee_fts_rename_file
gpFileIO_Write_File	qsee_fts_write_file

In addition to the interface renaming, the `gpHandle` parameter is removed from the new interfaces as it is not needed. For more details regarding the use of these interfaces on an older release, See *Fast Trusted Storage Feature Overview* (80-NM328-127).

2.3.3 Deprecated APIs

The following public APIs are deprecated in QTEE 5.0 but were previously available in QSEE 4.0. Also, various unpublished private APIs are deprecated or made unavailable.

- dbg_open
- file_end
- qsee_oem_process_cmd
- qsee_read_hdcp2p2_min_encryption_level

- qsee_read_register
- qsee_SW_Hash
- qsee_wait
- qsee_write_hdcp1_key
- qsee_write_hdcp2p2_key_riv_enable
- qsee_write_hdcp2p2_min_encryption_level
- QSEE_LOG

2.3.4 Deprecated system calls

Numerous undocumented native system calls are deprecated. Direct use of this system calls now results in an unsupported failure. Replace use of such system calls with the equivalent QTEE API to avoid future binary or source incompatibility.

2.3.5 Standard library APIs

Due to improved compatibility changes in QTEE 5.0, many C standard library functions are removed from the default QTEE libraries linked against TAs. QSEE 4.0 TAs that were dependent on the QTEE libraries for these removed C standard interfaces are likely to encounter linking errors when ported to QTEE 5.0. These errors are resolved by either updating the TA build files to link against the LLVM distributed version of the C standard library or by implementing the missing C standard functionality as part of the TA itself.

Although it is recommended that TAs can replace C standard library interfaces with those in the LLVM supplied C standard library, it is not recommended to use any beyond this replacement. Most of the C standard library APIs depend on system functionality that is not supported by QTEE and there is no documentation on which APIs are expected to work.

The QTEE libraries continue to support C standard library interfaces in cases where the LLVM C standard library implementation is insufficient. In QTEE 5.0, support for the C standard `malloc`, `realloc`, and `free` memory allocation interfaces are still available and also the file I/O interface documented in the supplied `qsee_fs.h` header.

2.4 Mink IPC interface updates

2.4.1 New IPC interfaces

The following new Mink IPC interface objects are added in QTEE 5.0.

2.4.2 IHlosRegionFinder

The `IHlosRegionFinder` interface object is added to allow TAs to create a shared memory region for accessing HLOS memory. The code sequence for acquiring the shared HLOS region object is as follows:

```
uint32_t ret;
Object regionFinder, memRegion;
/* Get an HLOS region finder interface object */
ret = qsee_open(CHlosRegionFinder_UID, &regionFinder);
if (Object_isERROR(ret)) {
  /* Handle error */
}
/* Acquire validated HLOS memory region interface object */
ret = IHlosRegionFinder_getRegion(regionFinder, pa, size, &memRegion);
if (Object_isERROR(ret)) {
  /* Handle error */
}
```

The following interfaces are provided from the `IHlosRegionFinder` interface object:

IHlosRegionFinder_release

Release a reference to the specified `IHlosRegionFinder` instance.

Syntax:

```
int32_t IHlosRegionFinder_release(Object self)
```

Parameters:

```
self – Object to release a reference
```

Returns:

```
Object_OK – Operation completed successfully
```

IHlosRegionFinder_retain

Retain a reference to the specified `IHlosRegionFinder` instance.

Syntax:

```
int32_t IHlosRegionFinder_retain(Object self)
```

Parameters:

```
self – Object to retain a reference
```

Returns:

```
Object_OK – Operation completed successfully
```

IHlosRegionFinder_getRegion

Acquire an HLOS memory region object described by the specified physical address and size.

Syntax:

```
int32_t IHlosRegionFinder_getRegion(Object self, uint64_t physAddr,
                                    uint64_t size, Object *memRegionOut)
```

Parameters:

```
self – IHlosRegionFinder object to invoke the operation
physAddr – Base physical address of the HLOS memory region to acquire
size – Size of the HLOS memory region to acquire
memRegionOut – Output pointer to the acquire memory region
```

Returns:

```
Object_OK – Operation completed successfully
Object_ERROR – Operation failed. Output values are undefined
```

2.4.2.1 IMemSpace

The `IMemSpace` interface object is added to allow TAs to map memory region objects to their address space. Mapping the HLOS memory object are coded as follows (continued from Section 2.4.2):

```
     Object space, memMap;
uintptr_t va;
size_t sizeout;
/* Get the application memory space object */
space = qsee_get_space();
if (Object_isNull(space)) {
  /* Handle error */
}
/* Map the HLOS memory object into the application for read access
 * Note: read-write access can be obtained using IMemRegion_PERM_RW.
 */
perm = IMemRegion_PERM_R;
ret = IMemSpace_map(space, memRegion, perm, &va, &sizeout, &memMap);
if (Object_isERROR(ret)) {
  /* Handle error */
}

/* Do read/write operations using output mapped 'va' */

/* Unmap region by releasing the 'memMap' object */
Object_RELEASE_IF(memMap);
Object_release(space);
```

The following interfaces are provided from the `IMemSpace` interface object:

IMemSpace_release

Release a reference to the specified `IMemSpace` instance.

Syntax:

```
int32_t IMemSpace_release(Object self)
```

Parameters:

```
self – Object to release a reference
```

Returns:

```
Object_OK – Operation completed successfully
```

IMemSpace_retain

Retain a reference to the specified `IMemSpace` instance.

Syntax:

```
int32_t IMemSpace_retain(Object self)
```

Parameters:

```
self – Object to retain a reference
```

Returns:

```
Object_OK – Operation completed successfully
```

IMemSpace_map

Map the specified memory object into the TA's address space.

Syntax:

```
int32_t IMemSpace_map(Object self, Object region, uint32_t perms,
                      uint64_t *virtAddr, uint64_t *sizeout,
                      Object *memMap)
```

Parameters:

```
self – IMemSpace object to invoke the operation
region – ImemRegion object to be mapped into the TA
perms – Permissions to use in mapping. Valid permissions include:
        IMemRegion_PERM_X – execute-only access
        IMemRegion_PERM_W – write-only access
        IMemRegion_PERM_R – read-only access
        IMemRegion_PERM_RW – read-write access
        IMemRegion_PERM_RX – read-execute access
virtAddr – Base virtual address of the mapped memory region
sizeout – Size of the mapped memory region
memMap – Output pointer to the memory mapping object
```

Returns:

```
Object_OK – Operation completed successfully
IMemSpace_ERROR_PERM – The requested mapping permissions are disallowed.
IMemSpace_ERROR_INVALID_REGION – The specified region is invalid
Object_ERROR – Operation failed. Output values are undefined
```

2.4.2.2 IMemRegion

The `IMemRegion` interface objects describe the properties of a given memory range available to the possessing TA and provides interfaces for operating on the memory range.

IMemRegion_release

Release a reference to the specified `IMemRegion` instance.

Syntax:

```
int32_t IMemRegion_release(Object self)
```

Parameters:

```
self – Object to release a reference
```

Returns:

```
Object_OK – Operation completed successfully
```

IMemRegion_retain

Retain a reference to the specified `IMemRegion` instance.

Syntax:

```
int32_t IMemRegion_retain(Object self)
```

Parameters:

```
self – Object to retain a reference
```

Returns:

```
Object_OK – Operation completed successfully
```

IMemRegion_getData

Read the data at the specified offset within the associated memory region into the output buffer.

Syntax:

```
int32_t IMemRegion_getData(Object self, uint64_t offset,
                           void *data_ptr, size_t data_len,
                           size_t *data_lenout)
```

Parameters:

```
self – IMemRegion object from which to read data
offset – Relative starting byte offset into the region to read
data_ptr – Pointer location of where to store the read data
data_len – Number of bytes to read from the specified offset
data_lenout – Pointer location of where to return the actual number
              of bytes read
```

Returns:

```
Object_OK – Operation completed successfully
Object_ERROR – Operation failed. Output values are undefined
```

IMemRegion_setData

Write the data at the provided memory buffer to the specified offset within the associated memory region.

Syntax:

```
int32_t IMemRegion_setData(Object self, uint64_t offset,
                           void *data_ptr, size_t data_len)
```

Parameters:

```
self – IMemRegion object in which to write data
offset – Relative starting byte offset into the region to write
data_ptr – Pointer location of source data to be written
data_len – Number of bytes to write from the specified offset
```

Returns:

```
Object_OK – Operation completed successfully
Object_ERROR – Operation failed. Output values are undefined
```

IMemRegion_createRestrictedRegion

Create a clone of the specified region with a restricted set of memory permissions.

Syntax:

```
int32_t IMemRegion_createRestrictedRegion(Object self, uint32_t perms,
                                          Object *region)
```

Parameters:

```
self – IMemRegion object to clone
perms – Restricted permission mask of the cloned region. Valid
        permissions include:
        IMemRegion_PERM_X – execute-only access
        IMemRegion_PERM_W – write-only access
        IMemRegion_PERM_R – read-only access
        IMemRegion_PERM_RW – read-write access
        IMemRegion_PERM_RX – read-execute access
region – Pointer location of where to return the resulting IMemRegion
         object.
```

Returns:

```
Object_OK – Operation completed successfully
Object_ERROR_KMEM – Operation failed due to memory allocation failure.
```

2.5 Header updates

2.5.1 New headers

The following new API headers are included in QTEE 5.0:

- `qsee_blsp.h` – BLSP (QUP v3) interfaces
- `qsee_cppf.h` – QTEE CPPF interfaces
- `qsee_drm.h` – QTEE DRM interfaces

- qsee_env.h – QTEE environment object interfaces
- qsee_error.h – QTEE error interfaces
- qsee_fts_crypto.h – QTEE fast trusted storage crypto interfaces
- qsee_fts_fileio.h – QTEE fast trusted storage file I/O interfaces
- qsee_hdcp.h – QTEE HDCP interfaces
- qsee_spcom.h – QTEE secure processor communication interfaces
- qsee_stor_rpmbw.h – QTEE RPMB storage interfaces
- qsee_uf_pbkdf2.h – QTEE password-based key derivation function interfaces

2.5.2 Modified headers

The following headers are updated in QTEE 5.0:

- gpPersistObjCrypto.h – Legacy FTS crypto interfaces made public
- gpPersistIbjFileIO.h – Legacy FTS file I/O interfaces made public
- qsee_bulletin_board.h – Increased TA name length
- qsee_cipher.h – Support added for SMMU virtual address cipher
- qsee_core.h – Component version APIs added
- qsee_ecc.h – Added affine point on curve interface
- qsee_ese_service.h – Numerous interface updates and additions
- qsee_fs.h – Formatting and comment changes, deprecated dbg_open
- qsee_fuse.h – Added/updated fuse enumerations
- qsee_i2c.h – Added new device IDs
- qsee_log.h – Deprecated QSEE_LOG macro
- qsee_macchiato.h – Added public signing key acquisition interface
- qsee_message.h – Added message TA name maximum length definition
- qsee_sec_camera.h – Numerous interface updates and additions
- qsee_services.h – Deprecated qsee_wait interface
- qsee_sfs.h – Deprecated unused error codes and decreased max filename size
- qsee_spi.h – Updated/added new device IDs and QUP v3 support
- qsee_stor.h – Added client removal and RPMB query enablement interfaces
- qsee_time.h – Added argument type definitions and time_end declaration
- qsee_tlmm.h – Updated to use standard integer types
- qsee_uf_aes.h – Added XTS mode support
- qsee_uf_sha.h – Deprecated qsee_SW_Hash interface, added type guards
- qsee_verstion.h – Updated QTEE version number definitions

2.5.3 Removed headers

The following API headers included in QSEE 4.0 are removed in QTEE 5.0:

- gpFileService.h
- gpPersistObjCommon.h
- gpPersistObjData.h
- gpPersistObjHandler.h
- gpPersistObjVersion.h
- gpObject.h
- gpPersistObjCrypto.h
- gpPersistObjFileIO.h
- gpPersistObjIndex.h
- qsee_alias.h
- qsee_ce_pipe.h
- qsee_counter.h
- qsee_ks.h
- qsee_ontario.h
- tzos_core_svc.h
- tzos_log.h
- tzos_scm.h
- tzos_scm_defs.h
- tzos_svc.h
- tzos_svc_common.h
- tzos_svc_def.h
- tzos_svc_id.h
- tzos_sys.h

Inclusion of these headers may result in compilation failures in QTEE 5.0.

A Reference

A.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
QTEE Version 5.0 User Guide	80-NH537-4
Fast Trusted Storage Feature Overview Application Note	80-NM328-127

A.2 Acronyms and terms

Acronym or term	Definition
FTS	Fast trusted storage
GP	Global platforms
SFS	Secure file system
QTEE	Qualcomm Trusted Execution Environment (formerly known as Qualcomm® secure execution environment (QSEE))