TRUSTED LITTLE KERNEL (TLK) FOR TEGRA: FOSS EDITION

PR_07297 | October 26, 2015 | Master Branch Release

References - Revision 2

Note: This page intentionally left blank.

Trusted Little Kernel

This Free Open Source Software (FOSS) release of the NVIDIA® Trusted Little Kernel (TLK) technology extends technology made available with the development of the Little Kernel (LK). You can download the LK modular embedded preemptive kernel for use on ARM®, x86, and AVR32 systems at:

`https://github.com/travisg/lk`

LK features include the following support:

- Cortex-M3, Cortex-A8, AVR32, and x86 SoC families
- Multi-threading, IPCs, Ext2 file system, and thread scheduling

Note: LK does not contain any TrustZone Technology. For information about TrustZone modes and architecture, see the whitepaper at:

`http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`

Earlier families of NVIDIA® Tegra® SoCs supported Trusted Foundations for security. Now NVIDIA implements its Trusted Little Kernel (TLK) technology, designed as a free, open-source trusted execution environment (OTE), for the following Tegra chipsets:

- NVIDIA® Tegra X1 families (this has a 64-bit architecture)
- NVIDIA® Tegra K1 64 Bit families
- NVIDIA® Tegra K1 32 Bit families
- NVIDIA® Tegra® 4i families

Trusted Little Kernel (TLK) technology is not limited to the Tegra chipset.

TLK features include:

- Small, pre-emptive kernel
- Support for multi-threading, IPCs, and thread scheduling
- Added TrustZone features
- Added Secure Storage
- Covered under MIT/FreeBSD license
- Support for GNU make

- Runs on most operating systems, including Android, Linux, and Windows.

NVIDIA extensions to Little Kernel (LK) include:

- User mode
- Address-space separation for TAs
- TLK Client Application (CA) library
- TLK TA library
- Adding user space (Bionic)
- Crypto library (encrypt/decrypt, key handling) via Open SSL
- Linux kernel driver (tlk_driver)
- Cortex A9/A15 support
- Power Management--Deep Sleep (LP0), Suspend (LP1), Standby (LP2)
- TrustZone memory carve-out (reconfigurable)
- Page table management
- On-the-fly (OTF) decoding driver for one-shot HW decode/encrypt
- Debugging support over UART (USB planned)

Overview

The host OS and NVIDIA® Trusted Little Kernel (TLK) software operate in a master-slave relationship; with TLK as the slave.

GNU make compiles and builds TLK. To facilitate debugging, TLK resides in a separate partition. The TLK current binary size is approximately 670 KB (with HDCP, etc.), with 2 MB of reconfigurable carve-out.

NVIDIA TLK has the following distinct environments and components:

- **Non-Secure Environment (NSE):** Operating system (OS) for the device running in normal mode.
- **Open Trusted Environment (OTE):** A separate, software-partitioned, environment that provides trusted operations.

  For example, OTE verifies that an application running in the OTE is not altered from its original, trusted condition.

- **Monitor:** Handles all communication between the NSE and OTE.
  - Activates the appropriate environment
  - Deactivates the appropriate environment
  - Routes function calls to the appropriate TLK libraries
  - Performs all environment switching

Switching occurs at the hardware level and is immediate. All applications running in the deactivated environment get suspended immediately.

- **Secure storage**: consists of several components that work together to store and access secured and encrypted data in a folder.
  - Components run inside and outside the OTE
  - Stored data can be generated by TLK itself or by a secure TA
  - Secure storage components manage security and encryption.

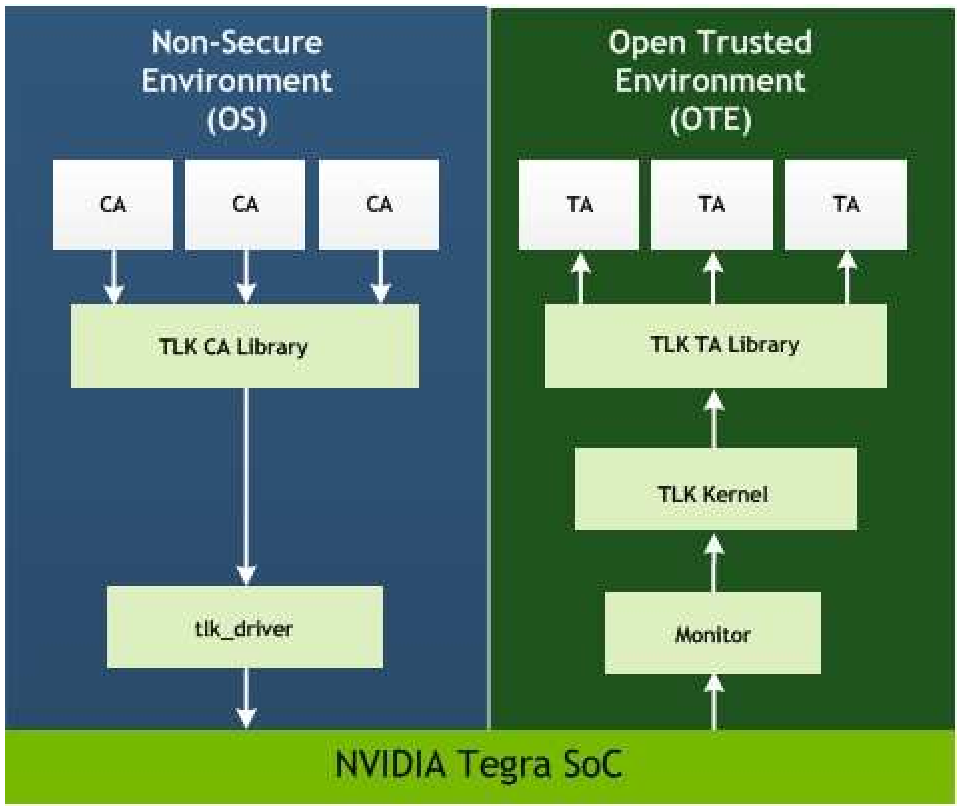
  The folder where the secure storage components save data provides a large storage area and is not restricted by the size of the TLK partition. Additionally, the folder backs up the partition. The `/data` partition is deleted upon factory reset and when obtaining root access with the `fastboot oem unlock` command.

  For more information, see Secure Storage in this chapter.

To use the OTE and secure storage, every **client application** (CA) running in the NSE (non-secure environment) **must** have a partner **trusted application** (TA) running in the secure world (OTE). The secure TA is the only component that can communicate with the secure storage service, as the secure storage service only accepts requests originating from a secure TA.

- All secure operations are initiated by a CA running in the NSE.
- A TA in the OTE never initiates contact with the NSE.

The following figure shows the relationships among the components.



Trusted Applications (TA) are post-linked at compile time, and TA properties are determined using the manifest. TAs have their own sandbox, and TA-to-TA communication is supported. Drivers are also considered TAs.

Execution Steps

1. When a client application (CA) needs a secure operation to be performed, it sends a request to a trusted application (TA) by invoking functions in the TLK CA library.
2. The TLK Driver routes the CA request to monitor, which routes the request through the TLK Kernel to the TLK TA Library.
3. The TLK TA library determines the appropriate TA to handle the request.
4. If the appropriate TA is not already running, the TLK TA library starts one.
5. The TLK TA library then passes control to the TA to handle the request.
6. Upon completion, execution control returns along the same path until reaching the CA, which receives a return value and any processed data.

Only one execution environment is active at any point in time. When the NSE is active, CAs can execute and the Monitor suspends TAs. Conversely, when the

Open Trusted Environment (OTE) is active, TAs can execute while the CAs are suspended.

TLK Libraries

The functions contained in the TLK CA and TLK TA libraries operate synchronously. Calls to these functions do not return until the requested operation has completed. You can find the TLK libraries at the following location:

```
$TOP/3rdparty/ote_partner/lib
```

Note:

The TLK TA libraries link to multiple libraries, including:

- TLK Interface library—a common interface for command execution.
- Bionic—the version found in the Android AOSP
- OpenSSL—the version found in the Android AOSP

The following table describes the TLK directories. Except for the tools directory, each is directory below corresponds to a public NVIDIA git project.

Directories	Description
3rdparty/ote_partner/tlk	TLK core source
3rdparty/ote_partner/lib	Library interfaces/client (API's) libs
tegra/ote_partner/tlk_driver	Linux driver between the NSE and Secure worlds. This source code is provided as an example only.
tegra/ote_partner/daemon	A proxy agent in the NSE world for TLK This source code is provided as an example only.
tegra/ote_partner/secure_monitor	A secure task (TA) that is responsible for the software running in the Monitor mode on ARM CPUs.
3rdparty/ote_partner/tasks	A secure task (TA).
3rdparty/ote_partner/tools	Toolchain to build TLK. You must independentally obtain these tools. For more information, see Obtaining the Toolchain in this chapter.

Secure Boot

NVIDIA® Trusted Little Kernel (TLK) software boots in the secure boot sequence.

BootROM:

1. Verifies the BCT and boot loader using PKC boot.
2. Jumps to the boot loader.

Boot loader:

3. Copies the Encrypted Key Storage (EKS) partition to TZ-SRAM.
4. Copies the TF_SBK to TZ-SRAM.
5. Verifies the TLK partition by using the PKC public key.
6. Jumps to the TLK partition with parameters such as the kernel boot address and TZ-DRAM parameters.

TLK:

7. Initializes the Monitor and sets up TZ-DRAM memory carve-out.
8. Copies the EKS partition from TZ-SRAM to TZ-DRAM.
9. Copies the TF_SBK to TZ-DRAM.
10. Sets the mode to non-secure and jumps to the kernel.

Linux kernel:

11. Boots the host OS.

Secure Storage

NVIDIA® Trusted Little Kernel (TLK) technology uses the `/data/tlk` folder on the host File system as the top-level secure storage directory. Secure storage files are placed in subdirectories under `/data/tlk` that use the secure storage client's UUID for their name. TLK encrypts and then stores secure data in files in these subdirectories where it can be subsequently decrypted and accessed.

A daemon running in non-secure space provides `/data/tlk` access to TLK. The TLK daemon performs the following basic file operations originating from the OTE:

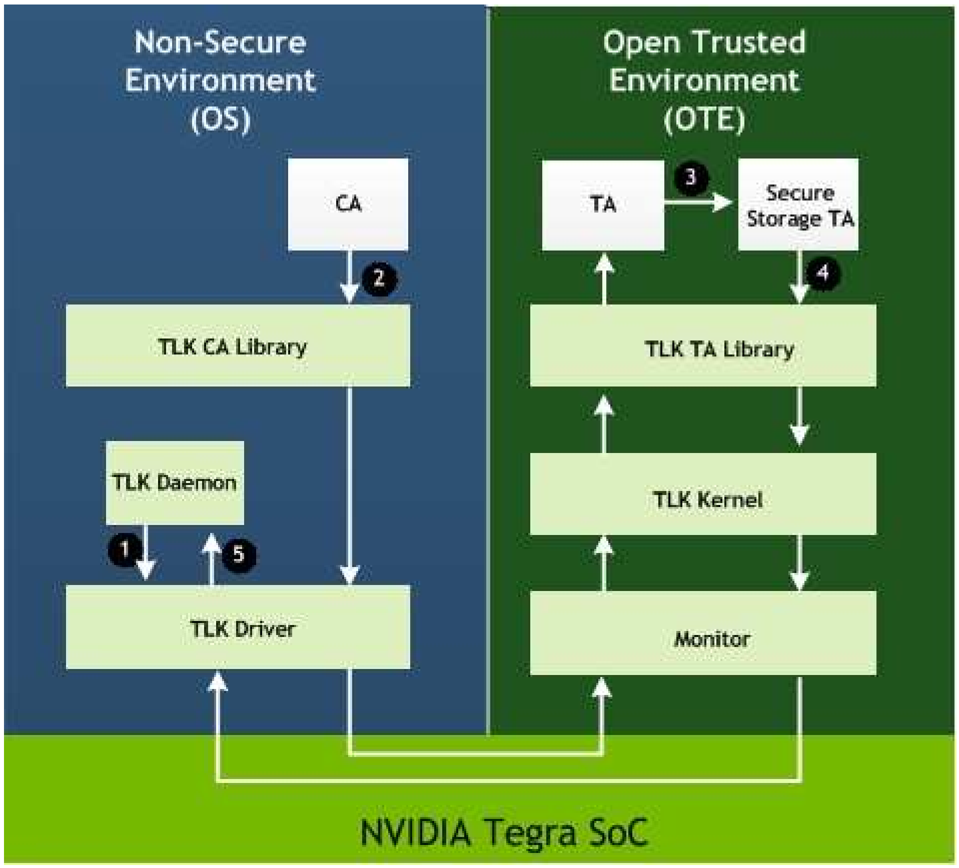
- create/delete
- open/close
- read/write
- getsize
- seek

- truncate

You can find the source code for the daemon here:

```
$TOP/3rdparty/ote_partner/daemon
```

Secure storage flow



The following table describes the numbered interactions in the diagram.

Interaction	Description
1	The TLK daemon is blocked on an IOCTL call waiting to service the next secure storage request.
2	A CA makes a secure storage request to a TA.
3	The TA sends the request to the Secure Storage TA.
4	The Secure Storage TA processes the request and forwards a packet to the TLK driver in the non-secure world.
5	The TLK driver notifies the TLK daemon that a new request is ready. The TLK daemon fetches the new request and issues the corresponding file operations and notifies the TLK driver that the request is complete.

The TLK daemon loads via the following initialization routine:

```
$TOP/device/nvidia/common/init.tlk.rc
```

The TLK daemon runs as user `system` and group `keystore`. During boot, the daemon starts with an input parameter of `--storagedir <dirname>`, which tells it which directory to use. This is a required parameter. Without it, the daemon will not boot.

The TLK daemon links with the `libtlk_client`, which provides APIs used to communicate with the Linux kernel. These APIs internally talk with the kernel via IOCTLs.

After boot, the TLK daemon makes an IOCTL call to the kernel to determine whether or not there is a new file operation request. If none, then the kernel blocks this IOCTL call until it has a new request.

Upon receiving a request from the TLK kernel, the TLK driver running in the Linux kernel packages the request for the TLK daemon and unblocks the daemon from inside the IOCTL call. Upon returning, the daemon parses the packet to determine and execute the corresponding file operations.

Create Operation

In create operations the daemon creates a file using the filename specified in the request. A file with that name must not already exist. The file is created under `/data/tlk/<uuid>`, where `<uuid>` is a text representation of the client TA's UUID.

Delete Operation

In delete operations the daemon deletes the file specified in the request.

Open Operation

In open operations the daemon opens the file specified in the request and returns a handle that can be used for subsequent read/write/seek/getsize/truncate operations. The open operation parameters include access flags for specifying read, write, or read-write access to the file.

Close Operation

In close operations the daemon closes access to the file using the handle from a previous open operation.

Read Operation

In read operations the daemon reads the specified number of bytes at the current file position using the specified handle. The handle must be generated in a previous open operation. The data is copied to the specified buffer for return to requester. The actual number of bytes read is returned to the caller.

Write Operation

In write operations the daemon writes the specified number of bytes at the current file position using the specified handle. The handle must be generated in a previous pen operation. The data written is copied from the specified buffer.

Get Size Operation

In getsize operations the daemon returns the current size in bytes of the file represented by the specified handle. The handle must be generated in a previous open operation.

Seek Operation

For seek operations the daemon sets the current position in the file represented by the specified handle to the specified offset. The handle must have been generated from a previous open operation.

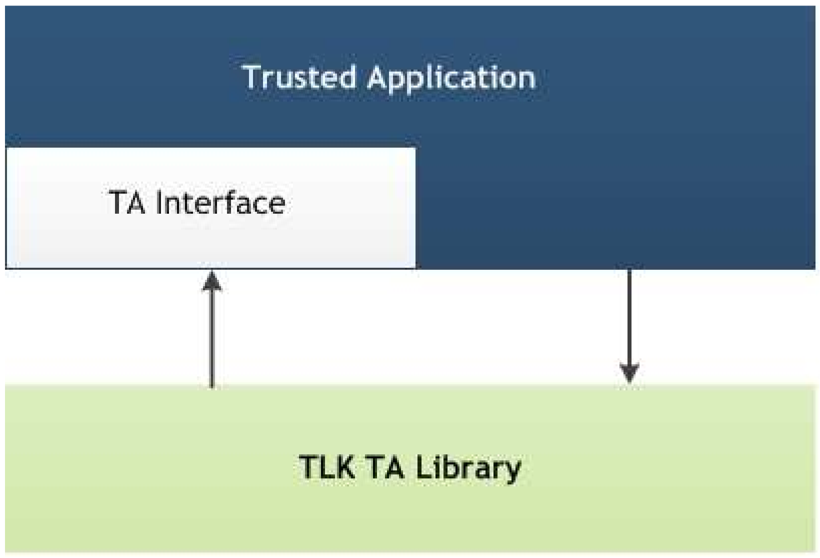
Truncate Operation

In truncate operations the daemon truncates the file represented by the handle to the specified length. The handle must be generated in a previous open operation. The specified length must be less than or equal the current size of the file.

TLK Trusted Application Development

A trusted application (TA) performs secure operations requested by a client application (CA). The CA request is routed to the TLK TA Library which calls the TA through a common function interface. This common TA interface is a group of functions that every TA must implement. The TLK TA Library also provides functions that a TA may call to perform data manipulation, cryptography, and other trusted operations.

The following diagram illustrates the relationships between a TA, its interface, and the TLK TA Library.



Installing and Building Secure Monitor

Before you can build a TLK device, you must install and build Secure Monitor for your Tegra SOC.

For prerequisites and `make` instructions, see the README at:

```
3rdparty/ote_partner/tlk
```

Secure Monitor NVIDIA Tegra X1

Secure Monitor for NVIDIA® Tegra® X1 is compiled as a standalone binary that runs by itself, independent of the Trusted OS (TLK) on the device. This binary runs in the EL3 exception mode on the CPU and implements mainly the PSCI interface and Fast SMC calls. TLK is dependent on this binary for its functioning and cannot run without a Monitor binary in place.

Tegra X1 CPUs support an NVIDIA monitor stack built that is provided as open-source. They also support the ARM Trusted Firmware stack.

The Trusted Firmware stack is a standard, open-source implementation provided by ARM. NVIDIA® supports the Trusted Firmware stack and strongly urges patrons to use it instead of the custom implementation. The Trusted Firmware contains a complete implementation of the PSCI specification and is guaranteed to contain multiple ARM v8 CPU fixes/erratas in the future. The NVIDIA custom implementation has the minimum code required to run TLK on Tegra. For more information on ARM Trusted Firmware, see:

`https://github.com/ARM-software/arm-trusted-firmware`

Secure Monitor for NVIDIA Tegra K1

NVIDIA® Tegra® K1 is compiled as a static library for ARMv7 CPUs, such as used in Tegra-K1. During compilation, TLK links with this library along with other components to generate the final image. The Secure Monitor is responsible for the software running in the Monitor mode on ARM CPUs. For now, this library implements the following features:

- PSCI interface

  http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0022c/index.html

- Fast SMC Calls, which are calls that are handled by the CPU's monitor mode without transitioning over to TLK
- Power management code, such as CPU ON, OFF, and SUSPEND

Building a TLK Device

This section describes how to build a TLK device. Although most of the information in this topic is specific for Android, TLK FOSS can be build and used for any OS. For prerequisites and `make` instructions, see the README at:

```
3rdparty/ote_partner/tlk
```

Required Components

- tlk—the secure Trusted Little Kernel (TLK).
- tasks—the secure tasks that run under the TLK.
- lib—the library support for tasks.
- client—library support for writing client applications that run under Linux in the non-secure-world. It also contains some example clients and tasks.
- Secure Monitor for your NVIDIA® Tegra® chip. This component is responsible for the software running in the Monitor mode on ARM CPUs.

Obtaining the Toolchain

Before you can build TLK, you must get the toolchain and populate the following directory:

```
3rdparty/ote_partner/tools
```

The required toolchain depends on your Tegra architecture:

- For 64-bit TLK: `tools/aarch64-linux-android-4.8`
- For 32-bit TLK: `tools/arm-eabi-4.7`

You can obtain these toolchains from:

https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.8
https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.7

Building a Task (Android)

A task is compiled as an ARM Elf Executable using a custom linker script. Below is the `Android.mk` for the Storage TA sample `test_app`.

Example Android Makefile

```
ifeq (tlk,$(SECURE_OS_BUILD))

LOCAL_PATH:= $(call my-dir)

## Make internal task
include $(NVIDIA_DEFAULTS)

LOCAL_MODULE_TAGS := optional
LOCAL_C_INCLUDES := $(LOCAL_PATH) \
             $(TOP)/3rdparty/ote_partner/lib/include
LOCAL_SRC_FILES := storagedemo.c manifest.c
LOCAL_MODULE := tlkstoragedemo_task
LOCAL_STATIC_LIBRARIES:= \
   libtlk_service \
   libtlk_common \
   libc

LOCAL_LDFLAGS := -T
$(TOP)/3rdparty/ote_partner/lib/task_linker_script.ld

LOCAL_FORCE_STATIC_EXECUTABLE := true
LOCAL_UNINSTALLABLE_MODULE := true
include $(NVIDIA_EXECUTABLE)

# Make client executable
include $(NVIDIA_DEFAULTS)
LOCAL_MODULE_TAGS := nvidia_tests
LOCAL_C_INCLUDES := $(LOCAL_PATH) \
         $(TOP)/3rdparty/ote_partner/lib/include \
         $(TEGRA_TOP)/core/include
LOCAL_SRC_FILES := client_storagedemo.c
LOCAL_STATIC_LIBRARIES := \
   libtlk_common \
   libtlk_client
LOCAL_SHARED_LIBRARIES := libnvos
LOCAL_MODULE := tlkstoragedemo_client

include $(NVIDIA_EXECUTABLE)

endif # SECURE_OS_BUILD == tlk
```

A module like this generates an executable at the following location:

```
$OUT/obj/EXECUTABLES/nvidia_tests_intermediates/tlkstoragedemo
_client
```

Building TLK

TLK builds after the tasks have been compiled. This is because the tasks are post-linked into the kernel to generate a complete image of TLK-plus-tasks. The dependencies in the `Android.mk` makefiles handle this ordering.

The `TASK_MODULES` variable in the `tlk/Android.mk` selects which tasks to post-link into the TLK. This list of modules is used to produce a list of executable files in:

```
$OUT/obj/EXECUTABLES/<task_module>_intermediates/<task_module>
```

The executable files get passed to the TLK make process.

TLK is built and post-linked with the executable files provided by the modules in `TASK_MODULES` to produce a `tos.img` file. This file is the complete TLK-plus-task image that will be flashed to the TOS partition on the device.

To build TLK

- In a terminal window, enter:

  ```
  cd tlk
  TARGET=<platform> make -e
  ```

  Where `<platform>` is `t132`.

Adding a Secure Service to a TLK Task (Android)

Adding a new task involves modifying the `makefile` located at:

```
$TOP/3rdparty/ote_partner/tlk/Android.mk
```

To Add a New Task

- Add the task name to the list of tasks in the TASK_MODULE variable of the `Android.mk` file.

The task name is the LOCAL_MODULE name for the task you wish to add. For example:

- For the crypto, the name is `tlkcryptodemo_task`
- For the storage sample, the name is `tlkstoragedemo_task`

- The same tasks are not added by default.
- Every time a task is updated; the `tos.img` must be regenerated and flashed onto the device.
- The client binaries, used to trigger an action in the task, are not built by default.
- The Android side binary trigger is performed through the adb shell.

To add a new task (Example)

1. On the host, open a shell and enter the following commands:

   ```
   mm
   adb remount
   adb sync
   ```

   The adb sync pushes the newly compiled binary to the device.

2. Invoke the binary to examine the adb shell.

3. Whenever the client is updated, push it to the device.

To flash the TLK

- To flash only the `tos.img` file use the following command:

  ```
  $ sudo $(which nvflash) –bl $OUT/bootloader.bin –download TOS
  $OUT/tos.img --go
  ```

  Note: The following command can only be used to trigger a TLK build:

  ```
  mp tlk
  ```

Manifests for Trusted Applications

Each Trusted Application (TA) must have a manifest that specifies various configuration properties for the TA. The manifest is described by the following data structure:

```
/*
 * Layout of .ote.manifest section in the trusted
 * application is the required UUID followed by an arbitrary
 * number of configuration options.
 */
```

```
typedef struct {
    te_service_id_t uuid;
    uint32_t configOptions[];
} OTE_MANIFEST;
```

Each TA must declare and initialize an instance of this `OTE_MANIFEST` structure. The data from that structure is placed in a well-defined section in the TA ELF binary where it can then be extracted and processed by TLK during TA initialization.

The following table lists the currently supported configuration options.

Configuration Option	Description
OTE_CONFIG_KEY_MIN_STACK_SIZE	Specifies the desired minimum stack size in bytes for use by TA tasks. The specified value is rounded up to the next page size (e.g., 4 K bytes). Use the `OTE_CONFIG_MIN_STACK_SIZE(sz)` macro to specify this option. The default for this option is one page (e.g., 4 K bytes).
OTE_CONFIG_KEY_MIN_HEAP_SIZE	Specifies the desired minimum heap size in bytes for use by TA tasks. The specified value is rounded up to the next page size (e.g., 4 K bytes). Use the `OTE_CONFIG_MIN_HEAP_SIZE(sz)` macro to specify this option. The default value for this option is four pages (e.g., 16 K bytes).
OTE_CONFIG_KEY_MAP_MEM	Enables the mapping of I/O regions owned by the TA to other (secure) TA clients. Use the `OTE_CONFIG_MAP_MEM(id,off,sz)` macro to enable these mappings. The `OTE_CONFIG_MAP_MEM(id,off,sz)` macro accepts these arguments: • id—specifies a unique 32-bit identifier for the region that is used as a handle when setting up a mapping. • off—specifies the 32-bit physical offset of the region. • sz—specifies the size in bytes of the region. The specified value is rounded up to the next page size (e.g., 4 K bytes).
OTE_CONFIG_RESTRICT_ACCESS (<what_to_block>)	Prevents certain types of clients from opening a connection or communicating with the TA. The macro takes an argument that describes which types of applications to block. To block CAs or non-secure applications:

	OTE_CONFIG_RESTRICT_ACCESS(OTE_RESTRICT_NON_SECURE_APPS)
	
	To block TAs but not CAs:
	OTE_CONFIG_RESTRICT_ACCESS(OTE_RESTRICT_SECURE_TASKS)
	
	To restrict all applications (no CAs or TAs):
	OTE_CONFIG_RESTRICT_ACCESS(OTE_RESTRICT_SECURE_TASKS | OTE_RESTRICT_NON_SECURE_APPS)

Example Manifest

The following shows the contents of an example manifest (`manifest.c`).

```
#include <service/ote_manifest.h>

OTE_MANIFEST OTE_MANIFEST_ATTRS ote_manifest =
{
    /* UUID : {32456890-8572-4a6f-a1f1-03aa9b05f9ff} */
    { 0x32456890, 0x8572, 0x4a6f,
        { 0xa1, 0xf1, 0x03, 0xaa, 0x9b, 0x05, 0xf9, 0xff } },

    /* optional configuration options here */
    {
        /* four pages for heap */
        OTE_CONFIG_MIN_HEAP_SIZE(4 * 4096),

    },
};
```

Legal Information

NVIDIA Legal Information

Open Source License

This NVIDIA product contain third party software that is being made available to you under its respective open source software license. That license requires specific legal information to be included in the product.

Little Kernel License

NVIDIA Tegra BSP for Android TLK FOSS API

Generated by NVIDIA

October 26, 2015

Contents

Chapter 1

Main Page

Welcome to *NVIDIA® Tegra® BSP for Android TLK Open Source API*. Documentation is preliminary and subject to change.

API Documentation

The API content of this document was generated from code comments. See the **Module Index** in the navigation pane for a list of the modules included in this documentation.

Warning

This API content represents the set of APIs you can **use directly**. Some APIs are not included in this document and we recommend you do not use them directly. Using undocumented APIs can lead to incompatibility when upgrading to later releases.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

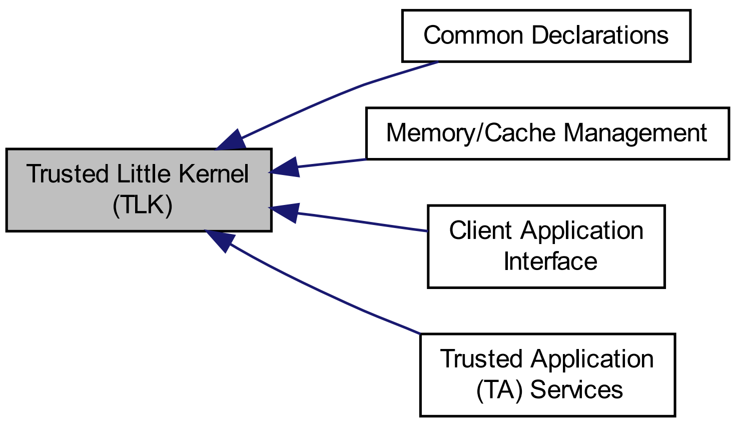
Module Documentation

5.1 Trusted Little Kernel (TLK)

5.1.1 Detailed Description

The NVIDIA® Trusted Little Kernel (TLK) technology extends technology made available with the development of the Little Kernel (LK). A TLK service is a Trusted Application (TA) that provides a service to other TAs. Examples of TLK services are: Crypto and OTF.

The Android OS and Trusted Little Kernel (TLK) software operate in a master-slave relationship, with TLK as the slave. TLK is compiled using the GCC when building your NVIDIA® Tegra® BSP for Android release. TLK resides in a separate partition, which facilitates debugging. Collaboration diagram for Trusted Little Kernel (TLK):


Common Declarations
Memory/Cache Management
Trusted Little Kernel (TLK)
Client Application Interface
Trusted Application (TA) Services

Modules

- Client Application Interface

  Defines the client application APIs.
- Common Declarations

  Defines the common declarations, functions, and error codes for the TLK.
- Memory/Cache Management

Defines TLK memory and cache management data types and functions.

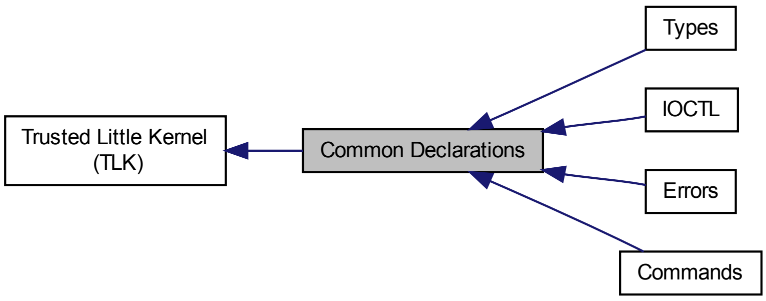
- Trusted Application (TA) Services

  Declarations and functions for the TA services.

5.2 Common Declarations

5.2.1 Detailed Description

Defines the common declarations, functions, and error codes for the TLK. Collaboration diagram for Common Declarations:



Modules

- Commands

    Defines common functions for Trusted Little Kernel (TLK).
- Errors

    Defines common error codes for Trusted Little Kernel (TLK).
- IOCTL

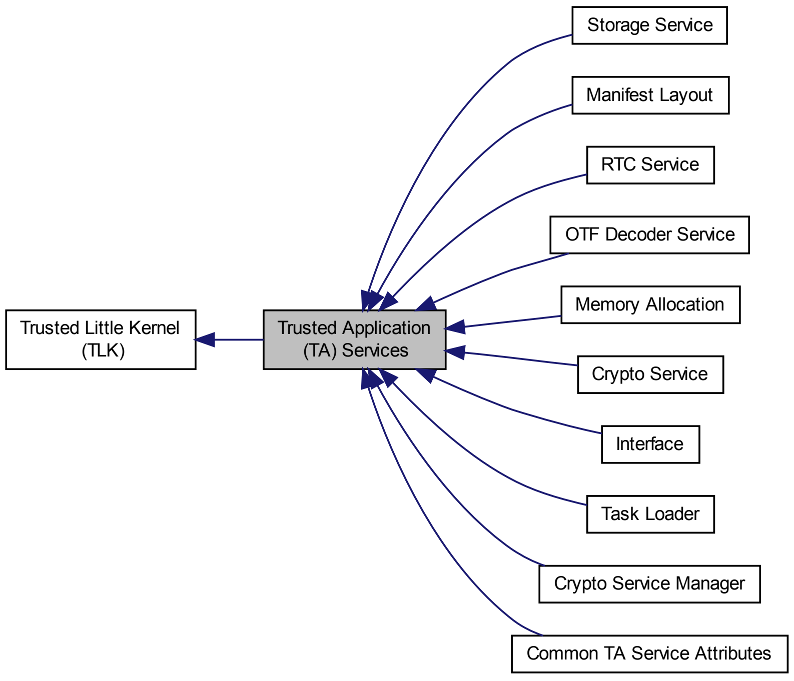
    Defines Trusted Little Kernel (TLK) IOCTL numbers for use by Trusted Application libraries.
- Types

    Defines common data types and functions for Trusted Little Kernel (TLK).

5.3 Trusted Application (TA) Services

5.3.1 Detailed Description

Declarations and functions for the TA services. Collaboration diagram for Trusted Application (TA) Services:



Modules

- Common TA Service Attributes

    Defines Trusted Little Kernel (TLK) common Trusted Application (TA) service attributes.
- Crypto Service

    Defines APIs for Trusted Little Kernel (TLK) crypto services.
- Crypto Service Manager

    Defines APIs for managing Trusted Little Kernel (TLK) crypto services.
- Interface

    Defines Trusted Application (TA) services declarations and functions.
- Manifest Layout

    Trusted Little Kernel (TLK) services manifest layout.
- Memory Allocation

    Defines Trusted Little Kernel (TLK) memory allocation services functions.
- OTF Decoder Service

    Defines Trusted Little Kernel (TLK) on-the-fly (OTF) decoder services functions.
- RTC Service

    Trusted Little Kernel (TLK) real-time clock (RTC) services.
- Storage Service

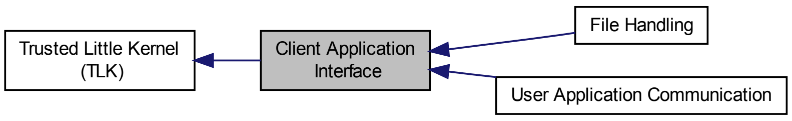
    Defines Trusted Little Kernel (TLK) storage services declarations and functions.
- Task Loader

Defines Trusted Application (TA) services declarations and functions for task loading and management.

5.4 Client Application Interface

5.4.1 Detailed Description

Defines the client application APIs. Collaboration diagram for Client Application Interface:


Trusted Little Kernel (TLK)
Client Application Interface
File Handling
User Application Communication

Modules

- File Handling
- User Application Communication

Macros

- #define TLK_DEVICE_BASE_NAME "tlk_device"
- #define TE_IOCTL_MAGIC_NUMBER ('t')
- #define TE_IOCTL_OPEN_CLIENT_SESSION _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x10, union te_cmd)
- #define TE_IOCTL_CLOSE_CLIENT_SESSION _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x11, union te_-cmd)
- #define TE_IOCTL_LAUNCH_OP _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x14, union te_cmd)
- #define TE_IOCTL_SS_CMD _IOR(TE_IOCTL_MAGIC_NUMBER, 0x30, int)
- #define TE_IOCTL_SS_CMD_GET_NEW_REQ 1
- #define TE_IOCTL_SS_CMD_REQ_COMPLETE 2

Enumerations

- enum {
  TLK_SMC_REQUEST = 0xFFFF1000,
  TLK_SMC_GET_MORE = 0xFFFF1001,
  TLK_SMC_ANSWER = 0xFFFF1002,
  TLK_SMC_NO_ANSWER = 0xFFFF1003,
  TLK_SMC_OPEN_SESSION = 0xFFFF1004,
  TLK_SMC_CLOSE_SESSION = 0xFFFF1005 }

  Defines secure monitor calls (SMC) that clients use to communicate with trusted applications (TAs) in the secure world.

5.4.2 Macro Definition Documentation

5.4.2.1 #define TLK_DEVICE_BASE_NAME "tlk_device"

5.4.2.2 #define TE_IOCTL_MAGIC_NUMBER ('t')

5.4.2.3 #define TE_IOCTL_OPEN_CLIENT_SESSION _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x10, union te_cmd)

5.4.2.4 #define TE_IOCTL_CLOSE_CLIENT_SESSION _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x11, union te_cmd)

5.4.2.5 #define TE_IOCTL_LAUNCH_OP _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x14, union te_cmd)

5.4.2.6 #define TE_IOCTL_SS_CMD _IOR(TE_IOCTL_MAGIC_NUMBER, 0x30, int)

5.4.2.7 #define TE_IOCTL_SS_CMD_GET_NEW_REQ 1

5.4.2.8 #define TE_IOCTL_SS_CMD_REQ_COMPLETE 2

5.4.3 Enumeration Type Documentation

5.4.3.1 anonymous enum

Defines secure monitor calls (SMC) that clients use to communicate with trusted applications (TAs) in the secure world.

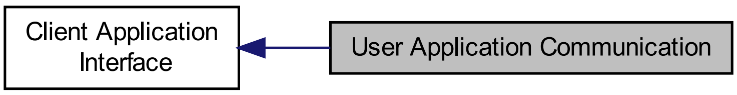
Enumerator:

TLK_SMC_REQUEST	Requests OTE to launch a TA operation.
TLK_SMC_GET_MORE	Gets a pending answer without making new operation.
TLK_SMC_ANSWER	Answers from secure side.
TLK_SMC_NO_ANSWER	No answers for now (secure side idle).
TLK_SMC_OPEN_SESSION	
TLK_SMC_CLOSE_SESSION	

5.5 User Application Communication

5.5.1 Detailed Description

Collaboration diagram for User Application Communication:



Data Structures

- struct te_answer
- struct ote_opensession

  Opens an open trusted environment (OTE) session.
- struct ote_closesession

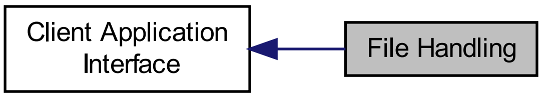
  Closes an OTE session.
- struct ote_launchop

  Launches an operation request.
- union te_cmd

5.6 File Handling

5.6.1 Detailed Description

Collaboration diagram for File Handling:



Data Structures

- struct ote_file_create_params_t
- struct ote_file_delete_params_t
- struct ote_file_open_params_t
- struct ote_file_close_params_t
- struct ote_file_write_params_t
- struct ote_file_read_params_t
- struct ote_file_seek_params_t
- struct ote_file_trunc_params_t
- struct ote_file_get_size_params_t
- struct ote_file_rpmb_write_params_t
- struct ote_file_rpmb_read_params_t
- union ote_file_req_params_t
- struct ote_file_req_t
- struct ote_ss_op_t

Macros

- #define OTE_MAX_DIR_NAME_LEN (64)
- #define OTE_MAX_FILE_NAME_LEN (128)
- #define OTE_MAX_DATA_SIZE (2048)
- #define OTE_RPMB_FRAME_SIZE 512
- #define SS_OP_MAX_DATA_SIZE 0x1000

Enumerations

- enum {
  OTE_FILE_REQ_TYPE_CREATE = 0x1,
  OTE_FILE_REQ_TYPE_DELETE = 0x2,
  OTE_FILE_REQ_TYPE_OPEN = 0x3,
  OTE_FILE_REQ_TYPE_CLOSE = 0x4,
  OTE_FILE_REQ_TYPE_READ = 0x5,
  OTE_FILE_REQ_TYPE_WRITE = 0x6,
  OTE_FILE_REQ_TYPE_GET_SIZE = 0x7,
  OTE_FILE_REQ_TYPE_SEEK = 0x8,
  OTE_FILE_REQ_TYPE_TRUNC = 0x9,
  OTE_FILE_REQ_TYPE_RPMB_WRITE = 0x1001,

  OTE_FILE_REQ_TYPE_RPMB_READ = 0x1002 }
- enum {
  OTE_FILE_REQ_FLAGS_ACCESS_RO = 1,
  OTE_FILE_REQ_FLAGS_ACCESS_WO = 2,
  OTE_FILE_REQ_FLAGS_ACCESS_RW = 3 }
- enum {
  OTE_SEEK_WHENCE_SET = 1,
  OTE_SEEK_WHENCE_CUR = 2,
  OTE_SEEK_WHENCE_END = 3 }

5.6.2 Macro Definition Documentation

5.6.2.1 #define OTE_MAX_DIR_NAME_LEN (64)

5.6.2.2 #define OTE_MAX_FILE_NAME_LEN (128)

5.6.2.3 #define OTE_MAX_DATA_SIZE (2048)

5.6.2.4 #define OTE_RPMB_FRAME_SIZE 512

5.6.2.5 #define SS_OP_MAX_DATA_SIZE 0x1000

5.6.3 Enumeration Type Documentation

5.6.3.1 anonymous enum

Enumerator:

OTE_FILE_REQ_TYPE_CREATE
OTE_FILE_REQ_TYPE_DELETE
OTE_FILE_REQ_TYPE_OPEN
OTE_FILE_REQ_TYPE_CLOSE
OTE_FILE_REQ_TYPE_READ
OTE_FILE_REQ_TYPE_WRITE
OTE_FILE_REQ_TYPE_GET_SIZE
OTE_FILE_REQ_TYPE_SEEK
OTE_FILE_REQ_TYPE_TRUNC
OTE_FILE_REQ_TYPE_RPMB_WRITE
OTE_FILE_REQ_TYPE_RPMB_READ

5.6.3.2 anonymous enum

Enumerator:

OTE_FILE_REQ_FLAGS_ACCESS_RO
OTE_FILE_REQ_FLAGS_ACCESS_WO
OTE_FILE_REQ_FLAGS_ACCESS_RW

5.6.3.3 anonymous enum

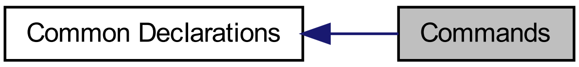
Enumerator:

OTE_SEEK_WHENCE_SET
OTE_SEEK_WHENCE_CUR
OTE_SEEK_WHENCE_END

5.7 Commands

5.7.1 Detailed Description

Defines common functions for Trusted Little Kernel (TLK). Collaboration diagram for Commands:



Functions

- te_error_t te_open_session (te_session_t ∗session, te_service_id_t ∗service, te_operation_t ∗operation)
- void te_close_session (te_session_t ∗session)
- te_operation_t ∗ te_create_operation (void)
- void te_init_operation (te_operation_t ∗te_op)
- void te_deinit_operation (te_operation_t ∗teOp)
- te_error_t te_launch_operation (te_session_t ∗session, te_operation_t ∗te_op)
- void te_oper_set_command (te_operation_t ∗te_op, uint32_t command)
- void te_oper_set_param_int_ro (te_operation_t ∗te_op, uint32_t index, uint32_t Int)
- void te_oper_set_param_int_rw (te_operation_t ∗te_op, uint32_t index, uint32_t Int)
- void te_oper_set_param_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗base, uint32_t len)
- void te_oper_set_param_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗base, uint32_t len)
- void te_oper_set_param_persist_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗base, uint32_t len)
- void te_oper_set_param_persist_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗base, uint32_t len)
- uint32_t te_oper_get_command (te_operation_t ∗te_op)
- uint32_t te_oper_get_num_params (te_operation_t ∗te_op)
- te_error_t te_oper_get_param_type (te_operation_t ∗te_op, uint32_t index, te_oper_param_type_t ∗type)
- te_error_t te_oper_get_param_int (te_operation_t ∗te_op, uint32_t index, uint32_t ∗Int)
- te_error_t te_oper_get_param_mem (te_operation_t ∗te_op, uint32_t index, void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_persist_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_persist_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗∗base, uint32-_t ∗len)
- void te_operation_reset (te_operation_t ∗te_op)

5.7.2 Function Documentation

5.7.2.1 te_error_t te_open_session (te_session_t ∗ *session,* te_service_id_t ∗ *service,* te_operation_t ∗ *operation*)

Opens a session to a TLK secure service.

Creates a session to a specified TLK secure service. This function must be called from the user side or from another TLK service.

Parameters

`out`	*session*	A pointer to a TLK secure service session to be initialized by this function.
`in`	*service*	The UUID/Service ID of the TLK service to which the caller wants to connect.
`in,out`	*operation*	A pointer to parameters that are expected by the TLK service.

Returns

OTE_SUCCESS Indicates the operation was successful.

5.7.2.2 void te_close_session (te_session_t ∗ *session*)

Closes an existing open session to a TLK secure service.

Parameters

`in`	*session*	A pointer to a TLK secure service session.

5.7.2.3 te_operation_t∗ te_create_operation (void)

Dynamically creates a new TLK secure service operation object.

This function creates a new `te_operation_t` object on the heap, and then initializes it.

Returns

A pointer to a TLK secure service operation object that was allocated on the stack and initialized.

5.7.2.4 void te_init_operation (te_operation_t ∗ *te_op*)

Initializes a TLK operation object.

This function initializes a TLK operation object using the TLK operation object pointer parameter passed to it. It is normally called to allocate a TLK operation object on the stack.

Parameters

`in,out`	*te_op*	A pointer to a TLK operation object.

5.7.2.5 void te_deinit_operation (te_operation_t ∗ *teOp*)

Deinitializes an existing TLK operation object.

Frees internal memory used by an existing TLK operation object.

Parameters

`in`	*teOp*	A pointer to a TLK secure service operation object.

5.7.2.6 te_error_t te_launch_operation (te_session_t ∗ *session*, te_operation_t ∗ *te_op*)

Sends an existing TLK operation object.

Sends the operation object to the TLK service. The operation object must have the command ID or necessary parameter setup.

Parameters

in	*session*	A pointer to an established TLK service session.
in,out	*te_op*	A te_operation_t object with proper parameters set up and a command ID.

Returns

The TLK secure service results after it services the command.

5.7.2.7 void te_oper_set_command (te_operation_t ∗ *te_op*, uint32_t *command*)

Sets a command to a TLK secure service operation object.

Sets the command ID to the operation object that will be sent to the TLK secure service.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*command*	A command ID that will be sent to the TLK secure service.

5.7.2.8 void te_oper_set_param_int_ro (te_operation_t ∗ *te_op*, uint32_t *index*, uint32_t *Int*)

Adds a read-only integer parameter to a TLK operation object.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*index*	An index value used to retrieve the parameter.
in	*Int*	An integer value.

5.7.2.9 void te_oper_set_param_int_rw (te_operation_t ∗ *te_op*, uint32_t *index*, uint32_t *Int*)

Adds a read-write integer parameter to a TLK operation object.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*index*	An index value used to retrieve the parameter.
in	*Int*	An integer value.

5.7.2.10 void te_oper_set_param_mem_ro (te_operation_t ∗ *te_op*, uint32_t *index*, const void ∗ *base*, uint32_t *len*)

Adds a read-only buffer parameter to the operation object. The buffer is the parameter at the given *index* in the object.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*index*	An index value used to retrieve the parameter.
in	*base*	The base address to the memory buffer.
in	*len*	The length of the memory buffer.

5.7.2.11 void te_oper_set_param_mem_rw (te_operation_t ∗ *te_op,* uint32_t *index,* void ∗ *base,* uint32_t *len*)

Adds a read-write buffer parameter to the operation object. The buffer is the parameter at the given *index* in the object.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`in`	*base*	The base address to the memory buffer.
`in`	*len*	The length of the memory buffer.

5.7.2.12 void te_oper_set_param_persist_mem_ro (te_operation_t ∗ *te_op,* uint32_t *index,* const void ∗ *base,* uint32_t *len*)

Adds a persistent read-only buffer parameter to the operation object. The buffer is the parameter at the given *index* in the object.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`in`	*base*	The base address to the memory buffer.
`in`	*len*	The length of the memory buffer.

5.7.2.13 void te_oper_set_param_persist_mem_rw (te_operation_t ∗ *te_op,* uint32_t *index,* void ∗ *base,* uint32_t *len*)

Adds a persistent read-write buffer parameter to the operation object. The buffer is the parameter at the given *index* in the object.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`in`	*base*	The base address to the memory buffer.
`in`	*len*	The length of the memory buffer.

5.7.2.14 uint32_t te_oper_get_command (te_operation_t ∗ *te_op*)

Gets a TLK command from an operation object.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.

Returns

The current command ID from the given operation object.

5.7.2.15 uint32_t te_oper_get_num_params (te_operation_t ∗ *te_op*)

Gets the number of parameters from an operation object.

Parameters

in	*te_op*	A pointer to a TLK operation object.

Returns

The number of parameters from the given operation object.

5.7.2.16 te_error_t te_oper_get_param_type (te_operation_t ∗ *te_op,* uint32_t *index,* te_oper_param_type_t ∗ *type*)

Gets the parameter type of a parameter.

Gets the parameter type of a parameter, given its index and operation object.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*index*	An index value used to retrieve the parameter.
out	*type*	The type of parameter.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *type* parameter was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *te_op* parameter was not found or the *type* parameter was not returned.

5.7.2.17 te_error_t te_oper_get_param_int (te_operation_t ∗ *te_op,* uint32_t *index,* uint32_t ∗ *Int*)

Gets an integer parameter from a given TLK operation object.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*index*	An index value used to retrieve the parameter.
out	*Int*	A pointer to an integer value.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *Int* parameter was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *index* was not found or the *Int* parameter was not an integer.

5.7.2.18 te_error_t te_oper_get_param_mem (te_operation_t ∗ *te_op,* uint32_t *index,* void ∗∗ *base,* uint32_t ∗ *len*)

Gets a memory buffer parameter from a given TLK operation object.

Parameters

in	*te_op*	A pointer to a TLK operation object.
in	*index*	An index value used to retrieve the parameter.
out	*base*	The base address of buffer.
out	*len*	The length of memory buffer.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *index* was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *index* was not found or the *te_op* parameter was not a memory buffer.

5.7.2.19 te_error_t te_oper_get_param_mem_ro (te_operation_t ∗ *te_op,* uint32_t *index,* const void ∗∗ *base,* uint32_t ∗ *len*)

Gets a read-only memory buffer parameter from given operation.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`out`	*base*	The base address of buffer.
`out`	*len*	The length of memory buffer.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *index* was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *index* was not found or the *te_op* parameter was not a memory buffer.

5.7.2.20 te_error_t te_oper_get_param_mem_rw (te_operation_t ∗ *te_op,* uint32_t *index,* void ∗∗ *base,* uint32_t ∗ *len*)

Gets a read/write memory buffer parameter from given operation.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`out`	*base*	The base address of buffer.
`out`	*len*	The length of memory buffer.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *index* was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *index* was not found or the *te_op* parameter was not a memory buffer.

5.7.2.21 te_error_t te_oper_get_param_persist_mem_ro (te_operation_t ∗ *te_op,* uint32_t *index,* const void ∗∗ *base,* uint32_t ∗ *len*)

Gets a read-only memory buffer parameter from given operation that persists until the session is closed.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`out`	*base*	The base address of buffer.
`out`	*len*	The length of memory buffer.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *index* was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *index* was not found or the *te_op* parameter was not a memory buffer.

5.7.2.22 te_error_t te_oper_get_param_persist_mem_rw (te_operation_t ∗ *te_op,* uint32_t *index,* void ∗∗ *base,* uint32_t ∗ *len*)

Gets a read/write memory buffer parameter from given operation that persists until the session is closed.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.
`in`	*index*	An index value used to retrieve the parameter.
`out`	*base*	The base address of buffer.
`out`	*len*	The length of memory buffer.

Return values

OTE_SUCCESS	Indicates the *te_op* parameter was found and the *index* was returned.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the *index* was not found or the *te_op* parameter was not a memory buffer.

5.7.2.23 void te_operation_reset (te_operation_t ∗ *te_op*)

Resets the data in an operation object.

When you call functions that set values within an operation object (for example te_oper_set_param_mem_rw()), the TLK interface allocates internal memory for those values. If you need to re-use an operation with new values, call `te_operation_reset`() before setting the new values.

Parameters

`in`	*te_op*	A pointer to a TLK operation object.

5.8 Types

5.8.1 Detailed Description

Defines common data types and functions for Trusted Little Kernel (TLK). Collaboration diagram for Types:

Common Declarations
Types

Data Structures

- struct te_service_id_t
- union te_session_t
- struct te_oper_param_t
- struct te_operation_t

Macros

- #define OTE_TASK_NAME_MAX_LENGTH 24
- #define OTE_TASK_PRIVATE_DATA_LENGTH 20

Typedefs

- typedef uint64_t cmnptr_t

Enumerations

- enum te_oper_param_type_t {
  TE_PARAM_TYPE_NONE = 0x0,
  TE_PARAM_TYPE_INT_RO = 0x1,
  TE_PARAM_TYPE_INT_RW = 0x2,
  TE_PARAM_TYPE_MEM_RO = 0x3,
  TE_PARAM_TYPE_MEM_RW = 0x4,
  TE_PARAM_TYPE_PERSIST_MEM_RO = 0x100,
  TE_PARAM_TYPE_PERSIST_MEM_RW = 0x101 }

Functions

- te_result_origin_t te_get_result_origin (te_session_t ∗session)

5.8.2 Macro Definition Documentation

5.8.2.1 #define OTE_TASK_NAME_MAX_LENGTH 24

Defines the maximum length of a zero-terminated informative task name.

5.8.2.2 #define OTE_TASK_PRIVATE_DATA_LENGTH 20

Defines the length of private data for the Trusted Application (TA). The definition goes in the manifest. The semantics of this optional data is defined per each TA.

To hold an SHA1 digest, this definition must be at least 20 bytes. Such a digest enables building a chain of trust to a TA with the manifest data.

This definition can be used to perform tasks such as:

- Loading public keys
- Loading X509 certificates and digests

5.8.3 Typedef Documentation

5.8.3.1 typedef uint64_t cmnptr_t

Holds a pointer large enough to support 32- and 64-bit clients.

5.8.4 Enumeration Type Documentation

5.8.4.1 enum te_oper_param_type_t

Specifies the operation object's parameter types.

Enumerator:

TE_PARAM_TYPE_NONE

TE_PARAM_TYPE_INT_RO

TE_PARAM_TYPE_INT_RW

TE_PARAM_TYPE_MEM_RO

TE_PARAM_TYPE_MEM_RW

TE_PARAM_TYPE_PERSIST_MEM_RO

TE_PARAM_TYPE_PERSIST_MEM_RW

5.8.5 Function Documentation

5.8.5.1 te_result_origin_t te_get_result_origin (te_session_t * *session*)

Returns the origin of a returned result.

Because it is possible for the operation to fail anywhere in the pipeline, this function returns the general block where the returned result originated.

Parameters

`in`	*session*	A valid session pointer.

Returns

A `te_result_origin_t` ID number.

5.9 Errors

5.9.1 Detailed Description

Defines common error codes for Trusted Little Kernel (TLK). Collaboration diagram for Errors:

Common Declarations
Errors

Enumerations

- enum te_error_t {
  OTE_SUCCESS = 0x00000000,
  OTE_ERROR_NO_ERROR = 0x00000000,
  OTE_ERROR_GENERIC = 0xFFFF0000,
  OTE_ERROR_ACCESS_DENIED = 0xFFFF0001,
  OTE_ERROR_CANCEL = 0xFFFF0002,
  OTE_ERROR_ACCESS_CONFLICT = 0xFFFF0003,
  OTE_ERROR_EXCESS_DATA = 0xFFFF0004,
  OTE_ERROR_BAD_FORMAT = 0xFFFF0005,
  OTE_ERROR_BAD_PARAMETERS = 0xFFFF0006,
  OTE_ERROR_BAD_STATE = 0xFFFF0007,
  OTE_ERROR_ITEM_NOT_FOUND = 0xFFFF0008,
  OTE_ERROR_NOT_IMPLEMENTED = 0xFFFF0009,
  OTE_ERROR_NOT_SUPPORTED = 0xFFFF000A,
  OTE_ERROR_NO_DATA = 0xFFFF000B,
  OTE_ERROR_OUT_OF_MEMORY = 0xFFFF000C,
  OTE_ERROR_BUSY = 0xFFFF000D,
  OTE_ERROR_COMMUNICATION = 0xFFFF000E,
  OTE_ERROR_SECURITY = 0xFFFF000F,
  OTE_ERROR_SHORT_BUFFER = 0xFFFF0010,
  OTE_ERROR_BLOCKED = 0xFFFF0011,
  OTE_ERROR_NO_ANSWER = 0xFFFF1003 }

  Defines Open Trusted Environment (OTE) error codes.
- enum te_result_origin_t {
  OTE_RESULT_ORIGIN_API = 1,
  OTE_RESULT_ORIGIN_COMMS = 2,
  OTE_RESULT_ORIGIN_KERNEL = 3,
  OTE_RESULT_ORIGIN_TRUSTED_APP = 4 }

  Defines the origin of an error.

5.9.2 Enumeration Type Documentation

5.9.2.1 enum te_error_t

Defines Open Trusted Environment (OTE) error codes.

Enumerator:

OTE_SUCCESS Indicates the operation was successful.

OTE_ERROR_NO_ERROR Indicates the operation was successful.

OTE_ERROR_GENERIC Indicates an unspecified error occurred.

OTE_ERROR_ACCESS_DENIED Indicates access privileges are insufficient.

OTE_ERROR_CANCEL Indicates the operation was cancelled.

OTE_ERROR_ACCESS_CONFLICT Indicates a concurrent accesses conflict.

OTE_ERROR_EXCESS_DATA Indicates data passed exceeds request.

OTE_ERROR_BAD_FORMAT Indicates input data is in an invalid format.

OTE_ERROR_BAD_PARAMETERS Indicates input parameters are invalid.

OTE_ERROR_BAD_STATE Indicates the operation is invalid in its current state.

OTE_ERROR_ITEM_NOT_FOUND Indicates the requested data item was not found.

OTE_ERROR_NOT_IMPLEMENTED Indicates the requested operation was not implemented.

OTE_ERROR_NOT_SUPPORTED Indicates the requested operation is not supported.

OTE_ERROR_NO_DATA Indicates the data expected is missing.

OTE_ERROR_OUT_OF_MEMORY Indicates the system ran out of resources.

OTE_ERROR_BUSY Indicates the system is busy.

OTE_ERROR_COMMUNICATION Indicates that communication failed.

OTE_ERROR_SECURITY Indicates a security fault was detected.

OTE_ERROR_SHORT_BUFFER Indicates the supplied buffer is too short.

OTE_ERROR_BLOCKED Indicates a task was administratively blocked; does not accept new sessions.

OTE_ERROR_NO_ANSWER Indicates no answer was received from the command target.

5.9.2.2 enum te_result_origin_t

Defines the origin of an error.

Enumerator:

OTE_RESULT_ORIGIN_API Indicates the error originated from the OTE Client API.

OTE_RESULT_ORIGIN_COMMS Indicates the error originated from the underlying communications stack.

OTE_RESULT_ORIGIN_KERNEL Indicates the error originated from the common OTE kernel code.

OTE_RESULT_ORIGIN_TRUSTED_APP Indicates the error originated from the Trusted Application code.

5.10 IOCTL

5.10.1 Detailed Description

Defines Trusted Little Kernel (TLK) IOCTL numbers for use by Trusted Application libraries. Collaboration diagram for IOCTL:

Common Declarations
IOCTL

Enumerations

- enum {
  OTE_IOCTL_GET_MAP_MEM_ADDR = 1UL,
  OTE_IOCTL_V_TO_P = 2UL,
  OTE_IOCTL_CACHE_MAINT = 3UL,
  OTE_IOCTL_TA_TO_TA_REQUEST = 4UL,
  OTE_IOCTL_GET_PROPERTY = 5UL,
  OTE_IOCTL_GET_DEVICE_ID = 6UL,
  OTE_IOCTL_GET_TIME_US = 7UL,
  OTE_IOCTL_GET_RAND32 = 8UL,
  OTE_IOCTL_SS_REQUEST = 9UL,
  OTE_IOCTL_TASK_REQUEST = 10UL,
  OTE_IOCTL_SS_GET_CONFIG = 11UL,
  OTE_IOCTL_REGISTER_TA_EVENT = 12UL,
  OTE_IOCTL_TASK_PANIC = 13UL }

  Defines IOCTL syscall interface parameters.

5.10.2 Enumeration Type Documentation

5.10.2.1 anonymous enum

Defines IOCTL syscall interface parameters.

Enumerator:

OTE_IOCTL_GET_MAP_MEM_ADDR

OTE_IOCTL_V_TO_P

OTE_IOCTL_CACHE_MAINT

OTE_IOCTL_TA_TO_TA_REQUEST

OTE_IOCTL_GET_PROPERTY

OTE_IOCTL_GET_DEVICE_ID

OTE_IOCTL_GET_TIME_US

OTE_IOCTL_GET_RAND32

OTE_IOCTL_SS_REQUEST

OTE_IOCTL_TASK_REQUEST

OTE_IOCTL_SS_GET_CONFIG

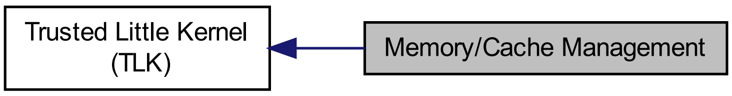
OTE_IOCTL_REGISTER_TA_EVENT

OTE_IOCTL_TASK_PANIC

5.11 Memory/Cache Management

5.11.1 Detailed Description

Defines TLK memory and cache management data types and functions. Collaboration diagram for Memory/Cache Management:



Data Structures

- struct te_map_mem_addr_args_t

  Holds a pointer to the map memory for a specific OTE_CONFIG_MAP_MEM ID value.
- struct te_v_to_p_args_t

  Holds a pointer to the physical address for a specific virtual address.
- struct te_cache_maint_args_t

  Holds an op code and data used to for cache maintenance.

Enumerations

- enum te_ext_nv_cache_maint_op_t {
  OTE_EXT_NV_CM_OP_CLEAN = 1,
  OTE_EXT_NV_CM_OP_INVALIDATE = 2,
  OTE_EXT_NV_CM_OP_FLUSH = 3 }
- enum te_ss_config_option_t {
  OTE_SS_CONFIG_RPMB_ENABLE = 0x0000001,
  OTE_SS_CONFIG_CPC_ENABLE = 0x0000002 }

Functions

- te_error_t te_ext_nv_cache_maint (te_ext_nv_cache_maint_op_t op, void ∗addr, uint32_t length)
- te_error_t te_ext_nv_virt_to_phys (void ∗addr, uint64_t ∗paddr)
- te_error_t te_ext_nv_get_map_addr (uint32_t id, void ∗∗addr)

5.11.2 Enumeration Type Documentation

5.11.2.1 enum te_ext_nv_cache_maint_op_t

Defines cache maintenance operations.

Enumerator:

OTE_EXT_NV_CM_OP_CLEAN	Cache clean operation.
OTE_EXT_NV_CM_OP_INVALIDATE	Cache invalidate operation.
OTE_EXT_NV_CM_OP_FLUSH	Cache flush operation.

5.11.2.2 enum te_ss_config_option_t

Defines a bit mask for secure storage configuration options.

Enumerator:

OTE_SS_CONFIG_RPMB_ENABLE	Defines the bit to enable RPMB rollback protection support.
OTE_SS_CONFIG_CPC_ENABLE	Defines the bit to enable CPC rollback protection support.

5.11.3 Function Documentation

5.11.3.1 te_error_t te_ext_nv_cache_maint (te_ext_nv_cache_maint_op_t *op*, void ∗ *addr*, uint32_t *length*)

Performs a cache maintenance operation.

This API allows the caller to perform a range of platform-specific cache management operations on the specified memory range.

Parameters

`in`	*op*	Cache maintenance operation to perform. See te_ext_nv_cache_maint_op_t for more information.
`in`	*addr*	Virtual address of the buffer on which the cache maintenance operation is to be performed.
`in`	*length*	Length in bytes of the buffer specified by the *addr* parameter.

Return values

OTE_SUCCESS	Indicates the operation completed successfully.
OTE_ERROR_BAD_PARAMETERS	Indicates one or more of the input parameters is invalid: • The value specified in the ID parameter is not a valid `te_ext_nv_cache_maint_op_t` value. • The virtual address specified in *addr* is not valid.
OTE_ERROR_OUT_OF_MEMORY	Indicates the system ran out of resources.
OTE_ERROR_GENERIC	Indicates an unspecified error occurred.

5.11.3.2 te_error_t te_ext_nv_virt_to_phys (void ∗ *addr*, uint64_t ∗ *paddr*)

Performs virtual-to-physical address translation.

This API returns the physical address that corresponds to the specified virtual adress.

Parameters

`in`	*addr*	The virtual address to translate.
`out`	*paddr*	A pointer with the physical address for *addr*.

Return values

OTE_SUCCESS	Indicates the operation completed successfully.
OTE_ERROR_BAD_PARAMETERS	Indicates one or more of the input parameters is invalid: • The virtual address specified in *addr* is not valid. • The *paddr* parameter contains an invalid pointer.

OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.
OTE_ERROR_GENERIC	Indicates an unspecified error occurred.

5.11.3.3 te_error_t te_ext_nv_get_map_addr (uint32_t *id*, void ** *addr*)

Retrieves mapping of a specified memory range.

This API returns the mapped address of the specified memory range setup in the service's manifest via the OTE_-CONFIG_MAP_MEM option.

Parameters

`in`	*id*	Memory region identifier specified in an `OTE_CONFIG_MAP_MEM` configuration option in the service's manifest.
`out`	*addr*	A pointer with the mapped address.

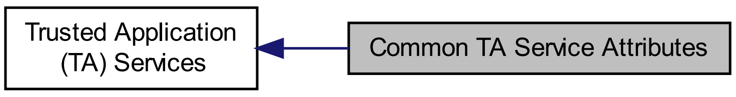
Return values

OTE_SUCCESS	Indicates the operation completed successfully.
OTE_ERROR_BAD_PARA-METERS	Indicates one or more of the input parameters is invalid: • The `OTE_CONFIG_MAP_MEM` ID parameter is unknown • The *addr* parameter contains an invalid pointer.
OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.
OTE_ERROR_GENERIC	Indicates an unspecified error occurred.

5.12 Common TA Service Attributes

5.12.1 Detailed Description

Defines Trusted Little Kernel (TLK) common Trusted Application (TA) service attributes. Collaboration diagram for Common TA Service Attributes:


Trusted Application (TA) Services
Common TA Service Attributes

Data Structures

- struct te_attribute_t

Macros

- #define OTE_ATTR_VAL 1 << 29
- #define OTE_ATTR_PUB 1 << 28

Enumerations

- enum te_attribute_id_t {
  OTE_ATTR_SECRET_VALUE = 0xC0000000,
  OTE_ATTR_RSA_MODULES = 0xD0000130,
  OTE_ATTR_RSA_PUBLIC_EXPONENT = 0xD0000230,
  OTE_ATTR_RSA_PRIVATE_EXPONENT = 0xC0000330,
  OTE_ATTR_RSA_PRIME1 = 0xC0000430,
  OTE_ATTR_RSA_PRIME2 = 0xC0000530,
  OTE_ATTR_RSA_EXPONENT1 = 0xC0000630,
  OTE_ATTR_RSA_EXPONENT2 = 0xC0000730,
  OTE_ATTR_RSA_COEFFICIENT = 0xC0000830,
  OTE_ATTR_DSA_PRIME = 0xD0001031,
  OTE_ATTR_DSA_SUBPRIME = 0xD0001131,
  OTE_ATTR_DSA_BASE = 0xD0001231,
  OTE_ATTR_DSA_PUBLIC_VALUE = 0xD0000131,
  OTE_ATTR_DSA_PRIVATE_VALUE = 0xD0000231,
  OTE_ATTR_DH_PRIME = 0xD0001032,
  OTE_ATTR_DH_SUBPRIME = 0xD0001132,
  OTE_ATTR_DH_BASE = 0xD0001232,
  OTE_ATTR_DH_X_BITS = 0xF0001332,
  OTE_ATTR_DH_PUBLIC_VALUE = 0xD0000132,
  OTE_ATTR_DH_PRIVATE_VALUE = 0xC0000232,
  OTE_ATTR_RSA_OAEP_LABEL = 0xD0000930,
  OTE_ATTR_RSA_PSS_SALT_LENGTH = 0xF0000A30 }

Functions

- te_error_t te_set_mem_attribute (te_attribute_t ∗attr, te_attribute_id_t id, void ∗buffer, uint32_t size)
- te_error_t te_get_mem_attribute_buffer (te_attribute_t ∗attr, void ∗∗ret)
- te_error_t te_get_mem_attribute_size (te_attribute_t ∗attr, size_t ∗ret)
- void te_copy_mem_attribute (void ∗buffer, te_attribute_t ∗key)
- te_error_t te_set_int_attribute (te_attribute_t ∗attr, te_attribute_id_t id, uint32_t a, uint32_t b)
- void te_free_internal_attribute (te_attribute_t ∗attr)
- te_error_t te_copy_attribute (te_attribute_t ∗dst, te_attribute_t ∗src)

5.12.2 Macro Definition Documentation

5.12.2.1 #define OTE_ATTR_VAL 1 << 29

Defines attribute ID type flag bitmasks.

5.12.2.2 #define OTE_ATTR_PUB 1 << 28

5.12.3 Enumeration Type Documentation

5.12.3.1 enum te_attribute_id_t

Defines attribute ID types.

Enumerator:

OTE_ATTR_SECRET_VALUE

OTE_ATTR_RSA_MODULES OTE_ATTR_PUB.

OTE_ATTR_RSA_PUBLIC_EXPONENT OTE_ATTR_PUB.

OTE_ATTR_RSA_PRIVATE_EXPONENT

OTE_ATTR_RSA_PRIME1

OTE_ATTR_RSA_PRIME2

OTE_ATTR_RSA_EXPONENT1

OTE_ATTR_RSA_EXPONENT2

OTE_ATTR_RSA_COEFFICIENT

OTE_ATTR_DSA_PRIME

OTE_ATTR_DSA_SUBPRIME OTE_ATTR_PUB.

OTE_ATTR_DSA_BASE OTE_ATTR_PUB.

OTE_ATTR_DSA_PUBLIC_VALUE OTE_ATTR_PUB.

OTE_ATTR_DSA_PRIVATE_VALUE OTE_ATTR_PUB.

OTE_ATTR_DH_PRIME OTE_ATTR_PUB.

OTE_ATTR_DH_SUBPRIME OTE_ATTR_PUB.

OTE_ATTR_DH_BASE OTE_ATTR_PUB.

OTE_ATTR_DH_X_BITS OTE_ATTR_VAL | OTE_ATTR_PUB.

OTE_ATTR_DH_PUBLIC_VALUE OTE_ATTR_PUB.

OTE_ATTR_DH_PRIVATE_VALUE

OTE_ATTR_RSA_OAEP_LABEL OTE_ATTR_PUB.

OTE_ATTR_RSA_PSS_SALT_LENGTH OTE_ATTR_VAL | OTE_ATTR_PUB.

5.12.4 Function Documentation

5.12.4.1 te_error_t te_set_mem_attribute (te_attribute_t * *attr*, te_attribute_id_t *id*, void * *buffer*, uint32_t *size*)

Sets up a memory attribute struct for use in other functions. This function stores the *id*, *buffer*, and *size* parameters in the memory attribute. The *id* parameter must be compatible with the memory attributes.

See Also

te_get_mem_attribute_buffer() and te_get_mem_attribute_size()

Parameters

`in,out`	*attr*	Attribute object.
`in`	*id*	Attribute ID of the buffer.
`in`	*buffer*	Memory location of the buffer.
`in`	*size*	Size of the buffer.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.12.4.2 te_error_t te_get_mem_attribute_buffer (te_attribute_t * *attr*, void ** *ret*)

Gets a memory attribute buffer. Stores in *ret* the address of the buffer in the memory attribute.

Parameters

`in`	*attr*	Attribute object.
`out`	*ret*	Return buffer pointer.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.12.4.3 te_error_t te_get_mem_attribute_size (te_attribute_t * *attr*, size_t * *ret*)

Gets memory attribute size. Stores in *ret* the size of the buffer in the mem attribute.

Parameters

`in`	*attr*	Attribute object.
`out`	*ret*	Return size.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.12.4.4 void te_copy_mem_attribute (void * *buffer*, te_attribute_t * *key*)

Copies the memory attribute. Copies the memory attribute buffer to the destination buffer, which must previously be allocated.

Parameters

`in,out`	*buffer*	Destination buffer.
`in`	*key*	Source attribute.

5.12.4.5 te_error_t te_set_int_attribute (te_attribute_t ∗ *attr,* te_attribute_id_t *id,* uint32_t *a,* uint32_t *b*)

Sets the integer attribute. Sets the attribute with integer values and *id*. The *id* parameter must match the integer type.

Return values

OTE_SUCCESS	Indicates the operation was successful.

Parameters

`in,out`	*attr*	Attribute object.
`in`	*id*	Attribute ID.
`in`	*a*	Integer value.
`in`	*b*	Integer value.

5.12.4.6 void te_free_internal_attribute (te_attribute_t ∗ *attr*)

Frees internal attribute memory. Frees any memory refereces by attribute.

Parameters

`in`	*attr*	Attribute object.

5.12.4.7 te_error_t te_copy_attribute (te_attribute_t ∗ *dst,* te_attribute_t ∗ *src*)

Copies attribute internals. After allocating space in destination attribute, this function copies integer memory attributes.

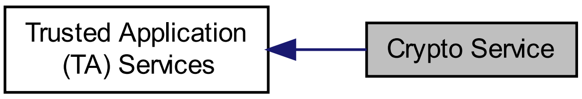
Parameters

`in,out`	*dst*	Destination attribute.
`in`	*src*	Source attribute.

5.13 Crypto Service

5.13.1 Detailed Description

Defines APIs for Trusted Little Kernel (TLK) crypto services. Collaboration diagram for Crypto Service:



Data Structures

- struct te_crypto_operation_info_t
- struct __te_crypto_operation_t
- struct te_crypto_rsa_key_t

Typedefs

- typedef struct __te_crypto_object ∗ te_crypto_object_t
- typedef struct __te_crypto_operation_t ∗ te_crypto_operation_t

Enumerations

- enum te_oper_crypto_algo_t {
  OTE_ALG_AES_ECB_NOPAD = 0x10000010,
  OTE_ALG_AES_CBC_NOPAD = 0x10000110,
  OTE_ALG_AES_CTR = 0x10000210,
  OTE_ALG_AES_CTS = 0x10000310,
  OTE_ALG_AES_ECB = 0x10000510,
  OTE_ALG_AES_CBC = 0x10000610,
  OTE_ALG_AES_CMAC_128 = 0x20000110,
  OTE_ALG_AES_CMAC_192 = 0x20000120,
  OTE_ALG_AES_CMAC_256 = 0x20000130,
  OTE_ALG_SHA_HMAC_224 = 0x20000210,
  OTE_ALG_SHA_HMAC_256 = 0x20000220,
  OTE_ALG_SHA_HMAC_384 = 0x20000230,
  OTE_ALG_SHA_HMAC_512 = 0x20000240,
  OTE_ALG_SHA_HMAC_1 = 0x20000250,
  OTE_ALG_RSA_PKCS_OAEP = 0x30000100,
  OTE_ALG_RSA_PSS = 0x30000200,
  OTE_ALG_PKCS1_Block1 = 0x30000300 }
- enum te_oper_crypto_algo_mode_t {
  OTE_ALG_MODE_ENCRYPT,
  OTE_ALG_MODE_DECRYPT,
  OTE_ALG_MODE_SIGN,
  OTE_ALG_MODE_VERIFY,
  OTE_ALG_MODE_DIGEST,

OTE_ALG_MODE_DERIVE }

Functions

- te_error_t te_allocate_object (te_crypto_object_t ∗obj)
- te_error_t te_populate_object (te_crypto_object_t obj, te_attribute_t ∗attrs, uint32_t attr_count)
- void te_free_object (te_crypto_object_t obj)
- te_error_t te_allocate_operation (te_crypto_operation_t ∗oper, te_oper_crypto_algo_t algorithm, te_oper_crypto_algo_mode_t mode)
- te_error_t te_set_operation_key (te_crypto_operation_t oper, te_crypto_object_t obj)
- te_error_t te_cipher_init (te_crypto_operation_t oper, void ∗iv, size_t iv_size)
- te_error_t te_cipher_update (te_crypto_operation_t oper, const void ∗src_data, size_t src_size, void ∗dst_data, size_t ∗dst_size)
- te_error_t te_cipher_do_final (te_crypto_operation_t oper, const void ∗src_data, size_t src_len, void ∗dst_data, size_t ∗dst_len)
- te_error_t te_rsa_init (te_crypto_operation_t oper)
- te_error_t te_rsa_handle_request (te_crypto_operation_t oper, const void ∗src_data, size_t src_size, void ∗dst_data, size_t ∗dst_size)
- void te_free_operation (te_crypto_operation_t oper)
- void te_generate_random (void ∗buffer, size_t size)
- te_error_t te_get_attribute_by_id (te_crypto_object_t object, te_attribute_id_t id, te_attribute_t ∗∗ret)

5.13.2 Typedef Documentation

5.13.2.1 typedef struct __te_crypto_object∗ **te_crypto_object_t**

5.13.2.2 typedef struct **__te_crypto_operation_t∗ te_crypto_operation_t**

5.13.3 Enumeration Type Documentation

5.13.3.1 enum **te_oper_crypto_algo_t**

Defines algorithm types.

Enumerator:

OTE_ALG_AES_ECB_NOPAD

OTE_ALG_AES_CBC_NOPAD

OTE_ALG_AES_CTR

OTE_ALG_AES_CTS

OTE_ALG_AES_ECB

OTE_ALG_AES_CBC

OTE_ALG_AES_CMAC_128

OTE_ALG_AES_CMAC_192

OTE_ALG_AES_CMAC_256

OTE_ALG_SHA_HMAC_224

OTE_ALG_SHA_HMAC_256

OTE_ALG_SHA_HMAC_384

OTE_ALG_SHA_HMAC_512

OTE_ALG_SHA_HMAC_1

OTE_ALG_RSA_PKCS_OAEP

OTE_ALG_RSA_PSS

OTE_ALG_PKCS1_Block1

5.13.3.2 enum te_oper_crypto_algo_mode_t

Defines algrorithm modes.

Enumerator:

OTE_ALG_MODE_ENCRYPT

OTE_ALG_MODE_DECRYPT

OTE_ALG_MODE_SIGN

OTE_ALG_MODE_VERIFY

OTE_ALG_MODE_DIGEST

OTE_ALG_MODE_DERIVE

5.13.4 Function Documentation

5.13.4.1 te_error_t te_allocate_object (te_crypto_object_t * *obj*)

Allocates memory for a te_crypto_object_t.

Parameters

`in,out`	*obj*	A pointer to new crypto object.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.

5.13.4.2 te_error_t te_populate_object (te_crypto_object_t *obj*, te_attribute_t * *attrs*, uint32_t *attr_count*)

Populates crypto object from a list of attributes. Allocates *obj* internal memory and copies attributes to *obj*.

Parameters

`in,out`	*obj*	Crypto object to store attributes.
`in`	*attrs*	Array of attributes.
`in`	*attr_count*	Array length.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.3 void te_free_object (te_crypto_object_t *obj*)

Frees allocated memory within crypto object.

Parameters

`in`	*obj*	Crypto object.

5.13.4.4 te_error_t te_allocate_operation (te_crypto_operation_t * *oper,* te_oper_crypto_algo_t *algorithm,* te_oper_crypto_algo_mode_t *mode*)

Allocates memory for crypto operation. Allocates crypto operation internal memory. Initializes operation based on algo/mode.

Parameters

`in,out`	*oper*	Crypto operation object.
`in`	*algorithm*	Crypto algorithm.
`in`	*mode*	Crypto algorithm mode.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.5 te_error_t te_set_operation_key (te_crypto_operation_t *oper,* te_crypto_object_t *obj*)

Allocates memory in the crypto operation and copies the key from the crypto object to the operation object.

Parameters

`in,out`	*oper*	Crypto operation object.
`in`	*obj*	Key crypto object.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.6 te_error_t te_cipher_init (te_crypto_operation_t *oper,* void * *iv,* size_t *iv_size*)

Initializes the operation cipher. Sets the initialization vector and calls the `init` function.

Parameters

`in,out`	*oper*	Crypto operation object.
`in`	*iv*	Initialization vector.
`in`	*iv_size*	Initialization vector size.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.7 te_error_t te_cipher_update (te_crypto_operation_t *oper,* const void * *src_data,* size_t *src_size,* void * *dst_data,* size_t * *dst_size*)

Updates the cipher by calling the operation `update` function with the supplied parameters.

Parameters

`in`	*oper*	Crypto operation object
`in`	*src_data*	Source data buffer supplied to `init`.
`in`	*src_size*	Source buffer size supplied to `init`.
`in,out`	*dst_data*	Destination data buffer supplied to `init`.
`in,out`	*dst_size*	Destination buffer size supplied to `init`.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.8 te_error_t te_cipher_do_final (te_crypto_operation_t *oper*, const void ∗ *src_data*, size_t *src_len*, void ∗ *dst_data*, size_t ∗ *dst_len*)

Calls operation `do_final` with supplied parameters.

Parameters

`in`	*oper*	Crypto operation object.
`in`	*src_data*	Source data buffer supplied to `do_final`.
`in`	*src_len*	Source buffer size supplied to `do_final`.
`in,out`	*dst_data*	Destination data buffer supplied to `do_final`.
`in,out`	*dst_len*	Destination buffer size supplied to `do_final`.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.9 te_error_t te_rsa_init (te_crypto_operation_t *oper*)

Initializes the RSA operation.

Parameters

`in,out`	*oper*	Crypto operation object.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.10 te_error_t te_rsa_handle_request (te_crypto_operation_t *oper*, const void ∗ *src_data*, size_t *src_size*, void ∗ *dst_data*, size_t ∗ *dst_size*)

Executes RSA operations.

Parameters

`in`	*oper*	Crypto operation object.
`in`	*src_data*	Source data buffer supplied to `init`.
`in`	*src_size*	Source buffer size supplied to `init`.
`in,out`	*dst_data*	Destination data buffer supplied to `init`.
`in,out`	*dst_size*	Destination buffer size supplied to `init`.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.13.4.11 void te_free_operation (te_crypto_operation_t *oper*)

Frees operation internal memory.

5.13.4.12 void te_generate_random (void ∗ *buffer,* size_t *size*)

Generates random data.

5.13.4.13 te_error_t te_get_attribute_by_id (te_crypto_object_t *object,* te_attribute_id_t *id,* te_attribute_t ∗∗ *ret*)

Finds the first attribute in the crypto object that matches ID.

Parameters

`in`	*object*	Crypto object.
`in`	*id*	Attribute ID to match.
`out`	*ret*	The attribute found.

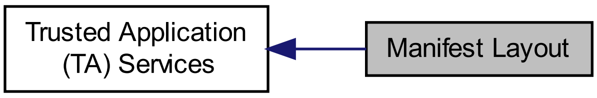
Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates the requested data item was not found.

5.14 Manifest Layout

5.14.1 Detailed Description

Trusted Little Kernel (TLK) services manifest layout. Collaboration diagram for Manifest Layout:



Data Structures

- struct OTE_MANIFEST

Macros

- #define OTE_CONFIG_MIN_STACK_SIZE(sz) OTE_CONFIG_KEY_MIN_STACK_SIZE, sz
- #define OTE_CONFIG_MIN_HEAP_SIZE(sz) OTE_CONFIG_KEY_MIN_HEAP_SIZE, sz
- #define OTE_CONFIG_MAP_MEM(id, off, sz) OTE_CONFIG_KEY_MAP_MEM, id, off, sz
- #define OTE_CONFIG_RESTRICT_ACCESS(clients) OTE_CONFIG_KEY_RESTRICT_ACCESS, clients
- #define OTE_CONFIG_AUTHORIZE(perm) OTE_CONFIG_KEY_AUTHORIZE, perm
- #define OTE_CONFIG_TASK_INITIAL_STATE(state) OTE_CONFIG_KEY_TASK_ISTATE, state
- #define OTE_MANIFEST_ATTRS __attribute((aligned(4))) __attribute((section(".ote.manifest")))

Enumerations

- enum ote_config_key_t {
  OTE_CONFIG_KEY_MIN_STACK_SIZE = 1,
  OTE_CONFIG_KEY_MIN_HEAP_SIZE = 2,
  OTE_CONFIG_KEY_MAP_MEM = 3,
  OTE_CONFIG_KEY_RESTRICT_ACCESS = 4,
  OTE_CONFIG_KEY_AUTHORIZE = 5,
  OTE_CONFIG_KEY_TASK_ISTATE = 6 }
- enum {
  OTE_RESTRICT_SECURE_TASKS = 1 << 0,
  OTE_RESTRICT_NON_SECURE_APPS = 1 << 1 }
- enum { OTE_AUTHORIZE_INSTALL = 1 << 10 }
- enum {
  OTE_MANIFEST_TASK_ISTATE_IMMUTABLE = 1 << 0,
  OTE_MANIFEST_TASK_ISTATE_STICKY = 1 << 1,
  OTE_MANIFEST_TASK_ISTATE_BLOCKED = 1 << 2 }

5.14.2 Macro Definition Documentation

5.14.2.1 #define OTE_CONFIG_MIN_STACK_SIZE(*sz*) OTE_CONFIG_KEY_MIN_STACK_SIZE, sz

Declares the minimum stack size.

Defines the minimum stack size the TLK service would expect.

Parameters

in	*sz*	The size of the stack in bytes.

5.14.2.2 #define OTE_CONFIG_MIN_HEAP_SIZE(*sz*) OTE_CONFIG_KEY_MIN_HEAP_SIZE, sz

Declares the minimum heap size.

Defines the minimum heap size the TLK service would expect.

Parameters

in	*sz*	The size of the heap in bytes.

5.14.2.3 #define OTE_CONFIG_MAP_MEM(*id, off, sz*) OTE_CONFIG_KEY_MAP_MEM, id, off, sz

Declares the memory address space needed.

Declares the physical memory address space the TLK service will require; a mapping will be created for TLK service.

Parameters

in	*id*	An ID number to later retrieve the mapping.
in	*off*	Base address of the physical address space.
in	*sz*	The size of the physical address space.

5.14.2.4 #define OTE_CONFIG_RESTRICT_ACCESS(*clients*) OTE_CONFIG_KEY_RESTRICT_ACCESS, clients

Declares client types that have restricted access.

Parameters

in	*clients*	A bit field to restrict access for client types.

5.14.2.5 #define OTE_CONFIG_AUTHORIZE(*perm*) OTE_CONFIG_KEY_AUTHORIZE, perm

Declares special actions that a TA is authorized to perform.

See Also

OTE_AUTHORIZE_INSTALL

Parameters

in	*perm*	A bit field that authorizes special actions for the TA.

5.14.2.6 #define OTE_CONFIG_TASK_INITIAL_STATE(*state*) OTE_CONFIG_KEY_TASK_ISTATE, state

Declares attributes for tasks, which apply beginning when the task is initially loaded.

See Also

OTE_MANIFEST_TASK_ISTATE_IMMUTABLE, OTE_MANIFEST_TASK_ISTATE_STICKY, and OTE_MANIFEST_TASK_ISTATE_BLOCKED.

Parameters

`in`	*state*	A bitfield to set the initial state to blocked.

5.14.2.7 #define OTE_MANIFEST_ATTRS __attribute((aligned(4))) __attribute((section(".ote.manifest")))

5.14.3 Enumeration Type Documentation

5.14.3.1 enum ote_config_key_t

Enumerator:

OTE_CONFIG_KEY_MIN_STACK_SIZE

OTE_CONFIG_KEY_MIN_HEAP_SIZE

OTE_CONFIG_KEY_MAP_MEM

OTE_CONFIG_KEY_RESTRICT_ACCESS

OTE_CONFIG_KEY_AUTHORIZE

OTE_CONFIG_KEY_TASK_ISTATE

5.14.3.2 anonymous enum

Defines bit flags for restricting task access by client type.

Use these bit flags with OTE_CONFIG_RESTRICT_ACCESS to restrict access for the type of client.

Enumerator:

OTE_RESTRICT_SECURE_TASKS

OTE_RESTRICT_NON_SECURE_APPS

5.14.3.3 anonymous enum

Defines special actions that the task can be authorized to perform. These actions are used with the OTE_CONFIG_AUTHORIZE macro. By default, the TA is not authorized for any special actions.

Currently, this enum defines only the `OTE_AUTHORIZE_INSTALL` bit. However, future releases may add support for other `OTE_AUTHORIZE_*` values. If that happens, you would set the OTE_CONFIG_AUTHORIZE *perm* argument with a bit field derived by ORing together the relevant `OTE_AUTHORIZE_*` values defined in this enum.

Enumerator:

OTE_AUTHORIZE_INSTALL Specifies the task is an installer.

5.14.3.4 anonymous enum

Defines bit flags that set attributes for the installed tasks. These values are used with the OTE_CONFIG_TASK_INITIAL_STATE macro. By default all attributes are unset.

To set a task's initial state attributes on load

- Set the OTE_CONFIG_KEY_TASK_ISTATE *state* argument with a bit field derived by ORing together the relevant `OTE_MANIFEST_TASK_ISTATE_*` values defined in this enum.

Enumerator:

OTE_MANIFEST_TASK_ISTATE_IMMUTABLE Task manifest cannot be modified by the installer.

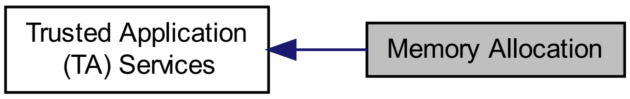
OTE_MANIFEST_TASK_ISTATE_STICKY Task cannot be unloaded.

OTE_MANIFEST_TASK_ISTATE_BLOCKED Task installed in BLOCKED state.

5.15 Memory Allocation

5.15.1 Detailed Description

Defines Trusted Little Kernel (TLK) memory allocation services functions. Collaboration diagram for Memory Allocation:



Functions

- void ∗ te_mem_alloc (uint32_t size)
- void ∗ te_mem_calloc (uint32_t size)
- void ∗ te_mem_realloc (void ∗buffer, uint32_t size)
- void te_mem_free (void ∗buffer)
- void te_mem_fill (void ∗buffer, uint32_t value, uint32_t size)
- void te_mem_move (void ∗dest, const void ∗src, uint32_t size)
- int te_mem_compare (const void ∗buffer1, const void ∗buffer2, uint32_t size)

5.15.2 Function Documentation

5.15.2.1 void∗ te_mem_alloc (uint32_t *size*)

Allocates memory of specified size.

Parameters

in	*size*	Size in bytes of desired memory.

Returns

A non-NULL pointer to the requested memory or NULL if the request to allocate memory failed.

5.15.2.2 void∗ te_mem_calloc (uint32_t *size*)

Allocates and clears memory of specified size.

Parameters

in	*size*	Size in bytes of desired memory.

Returns

A non-NULL pointer to the requested memory or NULL if the request to allocate and clear memory failed.

5.15.2.3 void∗ te_mem_realloc (void ∗ *buffer,* uint32_t *size*)

Changes size of memory by specified amount.

Parameters

`in`	*buffer*	A pointer to memory block to resize.
`in`	*size*	Specifies new size in bytes of memory block.

Returns

A non-NULL pointer to the newly resized memory or NULL if the request to resize failed.

5.15.2.4 void te_mem_free (void ∗ *buffer*)

Frees the specified memory.

Parameters

`in`	*buffer*	A pointer to memory to free.

5.15.2.5 void te_mem_fill (void ∗ *buffer,* uint32_t *value,* uint32_t *size*)

Fills specified memory range with specified value.

Parameters

`in`	*buffer*	A pointer to memory to fill.
`in`	*value*	Value to use for fill operation.
`in`	*size*	Number of bytes to fill.

5.15.2.6 void te_mem_move (void ∗ *dest,* const void ∗ *src,* uint32_t *size*)

Moves specified number of bytes from a source memory range to a destination memory range.

Note

The two memory ranges may overlap.

Parameters

`out`	*dest*	A pointer to destination buffer for move operation.
`in`	*src*	A pointer to source buffer for move operation.
`in`	*size*	Number of bytes to move.

5.15.2.7 int te_mem_compare (const void ∗ *buffer1,* const void ∗ *buffer2,* uint32_t *size*)

Compares two ranges of memory for equality.

Parameters

`in`	*buffer1*	A pointer to first memory range to compare.
`in`	*buffer2*	A pointer to second memory range to compare.
`in`	*size*	Number of bytes to compare.

Returns

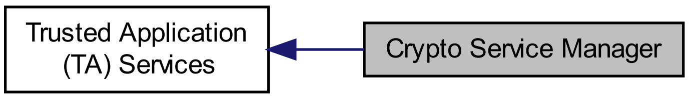
An integer where:

- 0 indicates the contents of the first memory range match match the contents of the second memory range.
- < 0 indicates the contents of the first memory range are less than the contents of the second memory range.
- > 0 indicates the contents of the first memory range are greater than the contents of the second memory range.

5.16 Crypto Service Manager

5.16.1 Detailed Description

Defines APIs for managing Trusted Little Kernel (TLK) crypto services. Collaboration diagram for Crypto Service Manager:



Functions

- te_error_t ote_nvcrypto_init (void)
- te_error_t ote_nvcrypto_deinit (void)
- te_error_t ote_nvcrypto_get_keybox (uint32_t keybox_index, void ∗buf, uint32_t ∗len)
- te_error_t ote_nvcrypto_get_storage_key (uint8_t ∗key, uint32_t key_size)

  Gets the storage key.
- te_error_t ote_nvcrypto_get_rollback_key (uint8_t ∗key, uint32_t key_size)

  Gets the rollback key.

5.16.2 Function Documentation

5.16.2.1 te_error_t ote_nvcrypto_init (void)

Initializes and opens an nvcrypto service session. This function keeps track of the number of open sessions.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.16.2.2 te_error_t ote_nvcrypto_deinit (void)

Closes an nvcrypto service session.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.16.2.3 te_error_t ote_nvcrypto_get_keybox (uint32_t *keybox_index,* void ∗ *buf,* uint32_t ∗ *len*)

Gets the keybox.

Parameters

`in`	*keybox_index*	The index of the keybox.
`in,out`	*buf*	A pointer to the key.
`in,out`	*len*	The length of the buffer in bytes.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.16.2.4 te_error_t ote_nvcrypto_get_storage_key (uint8_t ∗ *key*, uint32_t *key_size*)

Gets the storage key.

Return values

OTE_SUCCESS	Indicates the operation was successful.

Parameters

`in,out`	*key*	A pointer to the key.
`in`	*key_size*	The length of the key in bytes.

5.16.2.5 te_error_t ote_nvcrypto_get_rollback_key (uint8_t ∗ *key*, uint32_t *key_size*)

Gets the rollback key.

Return values

OTE_SUCCESS	Indicates the operation was successful.

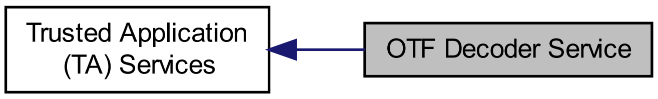
Parameters

`out`	*key*	A pointer to the key.
`in,out`	*key_size*	The length of the key in bytes.

5.17 OTF Decoder Service

5.17.1 Detailed Description

Defines Trusted Little Kernel (TLK) on-the-fly (OTF) decoder services functions. Collaboration diagram for OTF Decoder Service:



Functions

- te_error_t ote_otf_init (te_session_t ∗∗otfSession)

  Initializes the on-the-fly (OTF) hardware.
- te_error_t ote_otf_deinit (te_session_t ∗∗otfSession)

  Resets the OTF hardware and erases any previous keys.
- te_error_t ote_otf_setkey (void ∗buffer, uint32_t len, uint32_t ∗keySlot, te_session_t ∗otfSession)

  Sets the key to be used by the OTF hardware.
- te_error_t ote_otf_set_key_at (void ∗buffer, uint32_t len, uint32_t keySlot, te_session_t ∗otfSession)

  Sets the key to be used by the OTF hardware at the specified slot.
- te_error_t ote_otf_erasekey (te_session_t ∗otfSession)

  Erases keys from the OTF hardware.

5.17.2 Function Documentation

5.17.2.1 te_error_t ote_otf_init (te_session_t ∗∗ *otfSession*)

Initializes the on-the-fly (OTF) hardware.

Parameters

otfSession	A pointer to the OTF session.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_BAD_STATE	Indicates the session object was invalid.
OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.
OTE_ERROR_COMMUNI-CATION	Indicates that communication failed.

5.17.2.2 te_error_t ote_otf_deinit (te_session_t ∗∗ *otfSession*)

Resets the OTF hardware and erases any previous keys.

Parameters

otfSession	A pointer to the OTF session.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_BAD_STATE	Indicates the session object was invalid.

5.17.2.3 te_error_t ote_otf_setkey (void ∗ *buffer,* uint32_t *len,* uint32_t ∗ *keySlot,* te_session_t ∗ *otfSession*)

Sets the key to be used by the OTF hardware.

Parameters

`in`	*buffer*	A pointer to the key.
`in`	*len*	The length of the buffer in bytes.
`in,out`	*keySlot*	A pointer to the key to be used.
`in`	*otfSession*	A pointer to the OTF session.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_BAD_STATE	Indicates the session object was invalid.
OTE_ERROR_BAD_PARA-METERS	Indicates input parameters were invalid.
OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.
OTE_ERROR_COMMUNI-CATION	Indicates that communication failed.

5.17.2.4 te_error_t ote_otf_set_key_at (void ∗ *buffer,* uint32_t *len,* uint32_t *keySlot,* te_session_t ∗ *otfSession*)

Sets the key to be used by the OTF hardware at the specified slot.

Parameters

`in`	*buffer*	A pointer to the key.
`in`	*len*	The length of the buffer in bytes.
`in`	*keySlot*	Key slot to be used.
`in`	*otfSession*	A pointer to the OTF session.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_BAD_STATE	Indicates the session object was invalid.
OTE_ERROR_BAD_PARA-METERS	Indicates input parameters were invalid.
OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.
OTE_ERROR_COMMUNI-CATION	Indicates that communication failed.

5.17.2.5 te_error_t ote_otf_erasekey (te_session_t ∗ *otfSession*)

Erases keys from the OTF hardware.

Parameters

otfSession	A pointer to the OTF session.

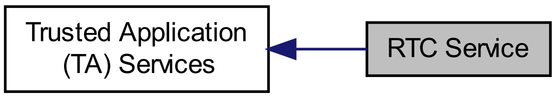
Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_BAD_STATE	Indicates the session object was invalid.
OTE_ERROR_OUT_OF_M-EMORY	Indicates the system ran out of resources.
OTE_ERROR_COMMUNI-CATION	Indicates that communication failed.

5.18 RTC Service

5.18.1 Detailed Description

Trusted Little Kernel (TLK) real-time clock (RTC) services. Collaboration diagram for RTC Service:



Functions

- te_error_t ote_rtc_init (void)

  Initializes the RTC hardware.
- te_error_t ote_rtc_deinit (void)

  Resets the RTC hardware and erases any previous keys.
- te_error_t ote_rtc_get_time (uint32_t ∗rtc)

  Gets the RTC from the RTC hardware.

5.18.2 Function Documentation

5.18.2.1 te_error_t ote_rtc_init (void)

Initializes the RTC hardware.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.18.2.2 te_error_t ote_rtc_deinit (void)

Resets the RTC hardware and erases any previous keys.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.18.2.3 te_error_t ote_rtc_get_time (uint32_t ∗ *rtc*)

Gets the RTC from the RTC hardware.

Parameters

out	*rtc*	A pointer to the RTC.

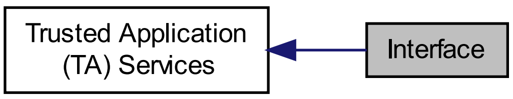
Return values

OTE_SUCCESS	Indicates the operation was successful.

5.19 Interface

5.19.1 Detailed Description

Defines Trusted Application (TA) services declarations and functions. Collaboration diagram for Interface:



Data Structures

- struct te_request_t

  Holds the layout of the `te_oper_param_t` structures which must match the layout sent in by the non-secure (NS) world via the TrustZone Secure Monitor Call (TZ SMC) path.
- struct te_ta_to_ta_request_args_t
- struct te_entry_point_message_t
- struct te_identity_t
- struct te_get_property_args_t
- struct te_device_unique_id
- struct te_panic_args_t
- struct ta_event_args_t

Macros

- #define DEVICE_UID_SIZE_BYTES 16
- #define OTE_PANIC_MSG_MAX_SIZE 128
- #define OTE_TE_FPRINTF_PREFIX_MAX_LENGTH (OTE_TASK_NAME_MAX_LENGTH + 4)

Typedefs

- typedef te_error_t(∗ ta_event_handler_t)(ta_event_args_t ∗args)

Enumerations

- enum {
  CREATE_INSTANCE = 1UL,
  DESTROY_INSTANCE = 2UL,
  OPEN_SESSION = 3UL,
  CLOSE_SESSION = 4UL,
  LAUNCH_OPERATION = 5UL,
  TA_EVENT = 6UL }
- enum {
  TE_LOGIN_PUBLIC = 0,
  TE_LOGIN_TA = 7 }

- enum {
  TE_PROP_DATA_TYPE_UUID = 1,
  TE_PROP_DATA_TYPE_IDENTITY = 2 }
- enum te_property_type_t {
  TE_PROPERTY_CURRENT_TA = 0xFFFFFFFF,
  TE_PROPERTY_CURRENT_CLIENT = 0xFFFFFFFE,
  TE_PROPERTY_TE_IMPLEMENTATION = 0xFFFFFFFD }
- enum ta_event_id {
  TA_EVENT_RESTORE_KEYS = 0,
  TA_EVENT_MASK = (1 << TA_EVENT_RESTORE_KEYS) }

Functions

- void te_exit_service (void)
- te_error_t te_init (int argc, char ∗∗argv)
- void te_destroy (void)
- te_error_t te_create_instance_iface (void)
- void te_destroy_instance_iface (void)
- te_error_t te_open_session_iface (void ∗∗sctx, te_operation_t ∗oper)
- void te_close_session_iface (void ∗sctx)
- te_error_t te_receive_operation_iface (void ∗sctx, te_operation_t ∗oper)
- void ∗ ote_get_instance_data (void)
- void ote_set_instance_data (void ∗sessionContext)
- te_error_t te_get_current_ta_uuid (te_service_id_t ∗value)
- te_error_t te_get_client_ta_identity (te_identity_t ∗value)
- te_error_t te_get_client_identity (te_identity_t ∗value)
- te_error_t te_get_device_unique_id (te_device_unique_id ∗uid)
- void te_panic (char ∗msg) __attribute__((noreturn))
- void te_fprintf_set_prefix (const char ∗prefix)
- void te_oper_dump_param (te_oper_param_t ∗param)
- void te_oper_dump_param_list (te_operation_t ∗te_op)
- te_error_t te_register_ta_event_handler (ta_event_handler_t handler, uint32_t events_mask)

5.19.2 Macro Definition Documentation

5.19.2.1 #define DEVICE_UID_SIZE_BYTES 16

5.19.2.2 #define OTE_PANIC_MSG_MAX_SIZE 128

Holds the panic information.

5.19.2.3 #define OTE_TE_FPRINTF_PREFIX_MAX_LENGTH (OTE_TASK_NAME_MAX_LENGTH + 4)

Defines the maximum length of the "[task_name] " prefix for the te_fprintf() task log entries.

5.19.3 Typedef Documentation

5.19.3.1 typedef te_error_t(∗ ta_event_handler_t)(ta_event_args_t ∗args)

5.19.4 Enumeration Type Documentation

5.19.4.1 anonymous enum

Enumerator:

- ***CREATE_INSTANCE***
- ***DESTROY_INSTANCE***
- ***OPEN_SESSION***
- ***CLOSE_SESSION***
- ***LAUNCH_OPERATION***
- ***TA_EVENT***

5.19.4.2 anonymous enum

Defines the supported login types.

Enumerator:

- ***TE_LOGIN_PUBLIC***
- ***TE_LOGIN_TA***

5.19.4.3 anonymous enum

Defines the type of property data.

Enumerator:

- ***TE_PROP_DATA_TYPE_UUID***
- ***TE_PROP_DATA_TYPE_IDENTITY***

5.19.4.4 enum te_property_type_t

Defines the property data information.

Enumerator:

- ***TE_PROPERTY_CURRENT_TA***
- ***TE_PROPERTY_CURRENT_CLIENT***
- ***TE_PROPERTY_TE_IMPLEMENTATION***

5.19.4.5 enum ta_event_id

Enumerator:

- ***TA_EVENT_RESTORE_KEYS***
- ***TA_EVENT_MASK***

5.19.5 Function Documentation

5.19.5.1 void te_exit_service (void)

5.19.5.2 te_error_t te_init (int *argc*, char ** *argv*)

Initializes the service.

5.19.5.3 void te_destroy (void)

Deinitializes the service.

5.19.5.4 te_error_t te_create_instance_iface (void)

Creates a new instance of the service.

5.19.5.5 void te_destroy_instance_iface (void)

Destroys an instance of the service.

5.19.5.6 te_error_t te_open_session_iface (void ** *sctx,* te_operation_t * *oper*)

Opens a session.

Parameters

sctx	A pointer to the session.
oper	A pointer to the operation.

5.19.5.7 void te_close_session_iface (void * *sctx*)

Closes an opened session.

Parameters

sctx	A pointer to the session to close.

5.19.5.8 te_error_t te_receive_operation_iface (void * *sctx,* te_operation_t * *oper*)

Receives an operation.

Parameters

sctx	A pointer to the session from which to receive the operation.
oper	A pointer to the operation.

5.19.5.9 void* ote_get_instance_data (void)

Gets the instance context data.

5.19.5.10 void ote_set_instance_data (void * *sessionContext*)

Sets an instance context data.

5.19.5.11 te_error_t te_get_current_ta_uuid (te_service_id_t * *value*)

Gets the service ID for the current Trusted Application (TA).

Parameters

out	*value*	A pointer to te_service_id_t, which holds the service ID.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.19.5.12 te_error_t te_get_client_ta_identity (te_identity_t ∗ *value*)

Gets the current client's identity only if it is a secure TA.

Parameters

out	*value*	A pointer to te_identity_t, which holds the client's identity.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.19.5.13 te_error_t te_get_client_identity (te_identity_t ∗ *value*)

Gets the current client's identity.

Parameters

out	*value*	A pointer to te_identity_t, which holds the client's identity.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.19.5.14 te_error_t te_get_device_unique_id (te_device_unique_id ∗ *uid*)

Gets the device unique ID.

Parameters

uid	A pointer to the device unique ID.

5.19.5.15 void te_panic (char ∗ *msg*)

Panics the system.

This call does not return.

Parameters

msg	A pointer to panic arguments.

5.19.5.16 void te_fprintf_set_prefix (const char ∗ *prefix*)

Set a printable prefix string that te_fprintf() outputs in front of every log message from this task.

The OTE library automatically sets a "[task_name] " log prefix based on the task name set in the task manifest (if the manifest defines a task name).

Parameters

in	*prefix*	The string to use for the prefix or NULL for no prefix. The maximum length of *prefix* is OTE_TE_FPRINTF_PREFIX_MAX_LENGTH. A NULL value cancels the log prefix; a non-null string changes the prefix.

5.19.5.17 void te_oper_dump_param (te_oper_param_t * *param*)

Prints the list of parameters for debugging.

Prints the list of parameters with the parameter content.

Parameters

in	*param*	A pointer to a TLK operation.

5.19.5.18 void te_oper_dump_param_list (te_operation_t * *te_op*)

Prints the list of parameters for debugging.

Prints the list of parameters with the parameter content.

Parameters

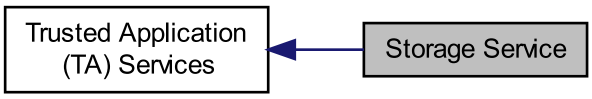
in	*te_op*	A pointer to a TLK operation.

5.19.5.19 te_error_t te_register_ta_event_handler (ta_event_handler_t *handler*, uint32_t *events_mask*)

5.20 Storage Service

5.20.1 Detailed Description

Defines Trusted Little Kernel (TLK) storage services declarations and functions. Collaboration diagram for Storage Service:



Macros

- #define TE_STORAGE_OBJID_MAX_LEN 64

Typedefs

- typedef struct
  __te_storage_object ∗ te_storage_object_t

Enumerations

- enum te_storage_flags_t {
  OTE_STORAGE_FLAG_ACCESS_READ = 0x1,
  OTE_STORAGE_FLAG_ACCESS_WRITE = 0x2,
  OTE_STORAGE_FLAG_ACCESS_WRITE_META = 0x4 }
- enum te_storage_whence_t {
  OTE_STORAGE_SEEK_WHENCE_SET = 0x1,
  OTE_STORAGE_SEEK_WHENCE_CUR = 0x2,
  OTE_STORAGE_SEEK_WHENCE_END = 0x3 }

Functions

- te_error_t te_create_storage_object (char ∗name, te_storage_flags_t flags, te_storage_object_t ∗obj)
- te_error_t te_open_storage_object (char ∗name, te_storage_flags_t flags, te_storage_object_t ∗obj)
- te_error_t te_read_storage_object (te_storage_object_t obj, void ∗buffer, uint32_t size, uint32_t ∗count)
- te_error_t te_write_storage_object (te_storage_object_t obj, const void ∗buffer, uint32_t size)
- te_error_t te_get_storage_object_size (te_storage_object_t obj, uint32_t ∗size)
- te_error_t te_seek_storage_object (te_storage_object_t obj, int32_t offset, te_storage_whence_t whence)
- te_error_t te_trunc_storage_object (te_storage_object_t obj, uint32_t size)
- te_error_t te_delete_storage_object (te_storage_object_t obj)
- te_error_t te_delete_named_storage_object (const char ∗name)
- te_error_t te_close_storage_object (te_storage_object_t obj)

5.20.2 Macro Definition Documentation

5.20.2.1 #define TE_STORAGE_OBJID_MAX_LEN 64

Defines the maximum file name length in bytes.

5.20.3 Typedef Documentation

5.20.3.1 typedef struct __te_storage_object∗ te_storage_object_t

5.20.4 Enumeration Type Documentation

5.20.4.1 enum te_storage_flags_t

Defines file access flags.

Enumerator:

OTE_STORAGE_FLAG_ACCESS_READ Specifies read access.

OTE_STORAGE_FLAG_ACCESS_WRITE Specifies write access.

OTE_STORAGE_FLAG_ACCESS_WRITE_META Specifies delete access.

5.20.4.2 enum te_storage_whence_t

Defines seek whence options.

Enumerator:

OTE_STORAGE_SEEK_WHENCE_SET Apply specified offset to start of file.

OTE_STORAGE_SEEK_WHENCE_CUR Apply specified offset to current position in file.

OTE_STORAGE_SEEK_WHENCE_END Apply specified offset to current end of file.

5.20.5 Function Documentation

5.20.5.1 te_error_t te_create_storage_object (char ∗ *name*, te_storage_flags_t *flags*, te_storage_object_t ∗ *obj*)

Creates a persistent storage object handle.

Precondition

This function must be called before accessing a file.

Parameters

in	*name*	Name of the persistent storage object (file).
in	*flags*	File access flags.
out	*obj*	A pointer to persistent object handle.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.2 te_error_t te_open_storage_object (char ∗ *name,* te_storage_flags_t *flags,* te_storage_object_t ∗ *obj*)

Opens a persistent storage object.

Precondition

This function must be called before accessing a file.

Parameters

in	*name*	Name of the persistent storage object (file).
in	*flags*	File access flags.
out	*obj*	A pointer to a persistent object handle.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.3 te_error_t te_read_storage_object (te_storage_object_t *obj,* void ∗ *buffer,* uint32_t *size,* uint32_t ∗ *count*)

Reads data from a persistent object.

The actual number of bytes read can be less than the requested value and does not necessarily mean a read failure.

Parameters

in	*obj*	Handle returned by te_open_storage_object().
in	*buffer*	Data buffer.
in	*size*	Size of the data buffer in bytes.
out	*count*	Actual number of bytes read.

Return values

OTE_SUCCESS	Indicates the operation was successful and *count* contains a non-zero value.

5.20.5.4 te_error_t te_write_storage_object (te_storage_object_t *obj,* const void ∗ *buffer,* uint32_t *size*)

Writes data to the persistent object.

Parameters

in	*obj*	Handle returned by te_open_storage_object().
in	*buffer*	Data buffer.
in	*size*	Size of the data buffer in bytes.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.5 te_error_t te_get_storage_object_size (te_storage_object_t *obj,* uint32_t ∗ *size*)

Gets the size of the data stored in the persistent object.

Parameters

`in`	*obj*	Handle returned by te_open_storage_object().
`in`	*size*	A pointer to hold the data size.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.6 te_error_t te_seek_storage_object (te_storage_object_t *obj*, int32_t *offset*, te_storage_whence_t *whence*)

Seeks to the specified offset in the persistent object.

Parameters

`in`	*obj*	Handle returned by te_open_storage_object().
`in`	*offset*	Number of bytes by which to adjust the data position. A positive value specifies to adjust the data position forward while a negative value specifies to adjust it backward.
`in`	*whence*	The position in the data from which to calculate the new position.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.7 te_error_t te_trunc_storage_object (te_storage_object_t *obj*, uint32_t *size*)

Truncates the data stored in the persistent object to the specified size. If the specified size is less than the current size then any residual data are lost.

Parameters

`in`	*obj*	Handle to writable persistent storage object returned by te_open_storage_-object().
`in`	*size*	The new size of the object's data in bytes.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.8 te_error_t te_delete_storage_object (te_storage_object_t *obj*)

Deletes a persistent object.

Parameters

`in`	*obj*	Handle returned by te_open_storage_object().

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.9 te_error_t te_delete_named_storage_object (const char * *name*)

Deletes a persistent object by name.

Parameters

`in`	*name*	Name of the persistent storage object (file).

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.20.5.10 te_error_t te_close_storage_object (te_storage_object_t *obj*)

Closes the persistent object handle.

This must be called when a client is done using the handle. On completion the handle will be invalid and te_open_-storage_object() must be used again to obtain a new handle.

Parameters

`in`	*obj*	Handle returned by `te_open_storage_object`().

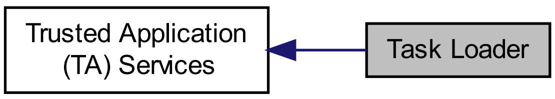
Return values

OTE_SUCCESS	Indicates the operation was successful.

5.21 Task Loader

5.21.1 Detailed Description

Defines Trusted Application (TA) services declarations and functions for task loading and management. The calls are restricted to tasks that have INSTALL permission set in the manifest. Calls from other tasks will be rejected. Collaboration diagram for Task Loader:



Data Structures

- struct te_app_load_memory_request_args_t
- struct te_task_info_t
- struct te_app_prepare_args_t
- struct te_task_restrictions_t
- struct te_app_start_args_t
- struct te_app_list_args_t
- struct te_list_apps
- struct te_get_task_info_t
- struct te_memory_mapping_t
- struct te_get_task_mapping_t
- struct te_get_pending_map_args_t
- struct te_system_info_args_t
- struct te_app_unload_args_t
- struct te_app_unload_t
- struct te_app_block_args_t
- struct te_app_block_t
- struct te_task_request_args_s

Typedefs

- typedef struct te_list_apps te_list_apps_t
- typedef struct
  te_task_request_args_s te_task_request_args_t

Enumerations

- enum install_type_t {
  INSTALL_TYPE_NORMAL = 0,
  INSTALL_TYPE_REJECT,
  INSTALL_TYPE_UPDATE }

  Defines install types for use with te_app_start() and te_app_start_args_t.

- enum te_get_info_type_t {
  OTE_GET_TASK_INFO_REQUEST_INDEX,
  OTE_GET_TASK_INFO_REQUEST_UUID,
  OTE_GET_TASK_INFO_REQUEST_SELF }
- enum te_app_id_t {
  OTE_APP_ID_INDEX,
  OTE_APP_ID_UUID }
- enum te_task_opcode_t {
  OTE_TASK_OP_UNKNOWN,
  OTE_TASK_OP_MEMORY_REQUEST,
  OTE_TASK_OP_PREPARE,
  OTE_TASK_OP_START,
  OTE_TASK_OP_LIST,
  OTE_TASK_OP_GET_TASK_INFO,
  OTE_TASK_OP_SYSTEM_INFO,
  OTE_TASK_OP_PENDING_MAPPING,
  OTE_TASK_OP_UNLOAD,
  OTE_TASK_OP_BLOCK,
  OTE_TASK_OP_UNBLOCK,
  OTE_TASK_OP_GET_MAPPING }

Functions

- te_error_t te_app_request_memory (u_int app_size, uintptr_t ∗app_addr, uint32_t ∗app_handle)
- te_error_t te_app_prepare (uint32_t app_handle, te_task_info_t ∗app_task_info)
- te_error_t te_app_start (uint32_t app_handle, install_type_t install_type, te_task_restrictions_t ∗app_-restrictions)
- te_error_t te_list_apps (te_list_apps_t ∗app_list)
- te_error_t te_task_get_info (te_get_task_info_t ∗task_info)
- te_error_t te_task_get_mapping (te_get_task_mapping_t ∗task_mapping)
- te_error_t te_get_pending_task_mapping (uint32_t app_handle, te_memory_mapping_t ∗app_map)
- te_error_t te_task_get_name_self (char ∗name, uint32_t ∗len_p)
- te_error_t te_task_system_info (uint32_t type)
- te_error_t te_app_unload (te_app_unload_t ∗arg_unload)
- te_error_t te_app_block (te_app_block_t ∗app)
- te_error_t te_app_unblock (te_app_block_t ∗app)

5.21.2 Typedef Documentation

5.21.2.1 typedef struct te_list_apps te_list_apps_t

Holds list command returned information.

5.21.2.2 typedef struct te_task_request_args_s te_task_request_args_t

Holds the task-loading ioctl argument used with OTE_IOCTL_TASK_REQUEST commands. The *te_task_request-_args_t::ia_opcode* field specifies the operation, which in turn indicates which argument from the union fields applies. For example, if *te_task_request_args_t::ia_opcode* is OTE_TASK_OP_MEMORY_REQUEST, then *te_task-_request_args_t::ia_load_memory_request* provides the memory request.

5.21.3 Enumeration Type Documentation

5.21.3.1 enum install_type_t

Defines install types for use with te_app_start() and te_app_start_args_t.

Enumerator:

INSTALL_TYPE_NORMAL Normal install operation.

INSTALL_TYPE_REJECT Reject (terminate install of a task.)

INSTALL_TYPE_UPDATE Replace a task if it already exists. Sticky tasks cannot be updated.

5.21.3.2 enum te_get_info_type_t

Type for the ioctl handler

Info request types:

Enumerator:

OTE_GET_TASK_INFO_REQUEST_INDEX

OTE_GET_TASK_INFO_REQUEST_UUID

OTE_GET_TASK_INFO_REQUEST_SELF

5.21.3.3 enum te_app_id_t

Enumerator:

OTE_APP_ID_INDEX

OTE_APP_ID_UUID

5.21.3.4 enum te_task_opcode_t

Defines opcodes for OTE_IOCTL_TASK_REQUEST ioctl argument for the above requests.

Enumerator:

OTE_TASK_OP_UNKNOWN

OTE_TASK_OP_MEMORY_REQUEST

OTE_TASK_OP_PREPARE

OTE_TASK_OP_START

OTE_TASK_OP_LIST

OTE_TASK_OP_GET_TASK_INFO

OTE_TASK_OP_SYSTEM_INFO

OTE_TASK_OP_PENDING_MAPPING

OTE_TASK_OP_UNLOAD

OTE_TASK_OP_BLOCK

OTE_TASK_OP_UNBLOCK

OTE_TASK_OP_GET_MAPPING

5.21.4 Function Documentation

5.21.4.1 te_error_t te_app_request_memory (u_int *app_size,* uintptr_t ∗ *app_addr,* uint32_t ∗ *app_handle*)

Allocates TLK kernel memory for task loading.

Parameters

in	*app_size*	Application size in bytes.
out	*app_addr*	Contains the task virtual address of the secure memory buffer allocated by TLK (where caller can write the application image to).
out	*app_handle*	Receives the opaque handle that can be used to manage app loading. Handle not valid after app loaded or an error occurs. The handle is a 32-bit random value set by TLK.

Return values

OTE_SUCCESS	Indicates the operation was successful.

Step 1/3 of task loading API.

5.21.4.2 te_error_t te_app_prepare (uint32_t *app_handle,* te_task_info_t ∗ *app_task_info*)

Requests the TLK to parse the static information from the task image loaded to the TLK shared memory and returns the task configuration information from the manifest section to the caller. The task is looked up by the APP_HANDLE received from the te_app_request_memory().

Parameters

in	*app_handle*	Handle for the copied application image to prepare.
out	*app_task_info*	Returns manifest data (see te_task_info_t) of the prepared task.

Return values

OTE_SUCCESS	Indicates the operation was successful.

Step 2/3 of task loading API.

5.21.4.3 te_error_t te_app_start (uint32_t *app_handle,* install_type_t *install_type,* te_task_restrictions_t ∗ *app_restrictions*)

Starts the loaded and prepared application.

Parameters

in	*app_handle*	Opaque handle for the application to start After this call returns the app_handle is no longer valid.
in	*install_type*	Specifies whether to reject the task, perform a normal task install, or update an installation. Updating an installation installs a new task in place of an old task and switches the old tasks's UUID to the new task. If the new task cannot be installed, the old task resumes operation.
in	*app_restrictions*	Specifies overrides for a subset of task properties in manifest (unless the manifest is declared immutable) or NULL for no overrides. Zero field values are ignored in overrides.

Return values

OTE_SUCCESS	Indicates the operation was successful.

Note

Currently this does not support "creating" a missing manifest for a task that does not have it in the binary ELF image, so also all loaded task images must contain a manifest.

Step 3/3 of task loading API.

5.21.4.4 te_error_t te_list_apps (te_list_apps_t * *app_list*)

Returns the UUID and optional name of current tasks by index.

Parameters

`in,out`	*app_list*	Indicates which task (0..N) UUID/NAME to look for.

Return values

OTE_SUCCESS	Indicates the operation was successful. OTE_ERROR_ITEM_NOT_FOUND indicates task at al_index is not valid.

al_name is a zero terminated possibly empty c-string.

5.21.4.5 te_error_t te_task_get_info (te_get_task_info_t * *task_info*)

Gets the task config info for any task in the system.

For loaded tasks the installer gets this info also with other means, but for static tasks (like the installer itself) this function gets the manifest information, particularly the private_data field which in the installer case configured the trust anchor (digest of root cert data blob).

This function also returns current task state info in the `gti_state` and `gti_type` fields of te_get_task_info_t.

Parameters

`in,out`	*task_info*	Request the manifest information of any task (static or loaded) gti_request_type and the union gtiu_* fields [in] identify the task and gti_info [out] contains the te_task_info_t response.

5.21.4.6 te_error_t te_task_get_mapping (te_get_task_mapping_t * *task_mapping*)

Gets the specified manifest address mapping from the task.

Parameters

`in,out`	*task_mapping*	A pointer to manifest mapping information of any task (static or loaded). The `gmt_request_type` and the union `gtiu_*` fields [in] identify the task and `gmt_map` field map_index specifies the mapping (0..N).

This function's response is in the *task_mapping* fields:

- `map_id` specifies to the mapping ID.
- `map_offset` specifies to the physical address to map.
- `map_size` specifies size of mapped area.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.21.4.7 te_error_t te_get_pending_task_mapping (uint32_t *app_handle,* te_memory_mapping_t ∗ *app_map*)

Gets the specified manifest address mapping from task by its installation handle before the task is installed (this is an installer only API).

Parameters

`in`	*app_handle*	The handle for the pending task.
`in,out`	*app_map*	A pointer to task mapping information. `app_map->map_index` is the index [0..N] of the fetched mapping.

This function's response is in the *app_map* fields:

- `map_id` specifies to the mapping ID.
- `map_offset` specifies to the physical address to map.
- `map_size` specifies size of mapped area.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.21.4.8 te_error_t te_task_get_name_self (char ∗ *name,* uint32_t ∗ *len_p*)

Gets the current task name from its own manifest record.

Parameters

`out`	*name*	Copy task name from manifest record.
`in,out`	*len_p*	Contains the max length of the name on input and the length of the copied manifest name.

Return values

OTE_SUCCESS	Indicates the operation was successful.

5.21.4.9 te_error_t te_task_system_info (uint32_t *type*)

Outputs task load system info to console, for test/debug purposes. DEBUG API only for information.

5.21.4.10 te_error_t te_app_unload (te_app_unload_t ∗ *arg_unload*)

Unloads a previously loaded application.

Parameters

`in`	*arg_unload*	Identifies the application to unload (XX now contains UUID)

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_ITEM_NOT-_FOUND	Indicates a task with the specified uuid did not exist.

5.21.4.11 te_error_t te_app_block (te_app_block_t * *app*)

Blocks a task.

Parameters

in	*app*	Identifies the application to block by INDEX or UUID.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_ITEM_NOT-_FOUND	indicates specified task was not found. + others...

5.21.4.12 te_error_t te_app_unblock (te_app_block_t * *app*)

Unblocks a previously blocked application.

Parameters

in	*app*	Identifies the application to unblock by INDEX or UUID.

Return values

OTE_SUCCESS	Indicates the operation was successful.
OTE_ERROR_ITEM_NOT-_FOUND	indicates specified task was not found. + others...

Chapter 6

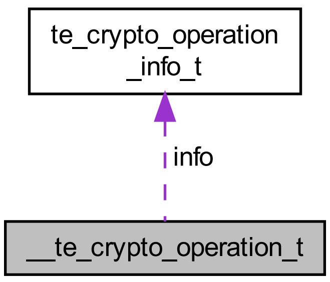
Data Structure Documentation

6.1 __te_crypto_operation_t Struct Reference

6.1.1 Detailed Description

Internal data structure for `te_crypto_operation_t`.

Collaboration diagram for __te_crypto_operation_t:



Data Fields

- te_crypto_operation_info_t info
- void ∗ key
- void ∗ iv
- size_t iv_len
- void ∗ imp_obj
- te_error_t(∗ init)(te_crypto_operation_t operation)
- te_error_t(∗ update)(te_crypto_operation_t operation, const void ∗src_data, size_t src_size, void ∗dst_dat, size_t ∗dst_size)
- te_error_t(∗ do_final)(te_crypto_operation_t operation, const void ∗srd_data, size_t src_size, void ∗dst_data, size_t ∗dst_size)
- te_error_t(∗ handle_req)(te_crypto_operation_t operation, const void ∗src_data, size_t src_size, void ∗dst_-data, size_t ∗dst_size)
- void(∗ free)(te_crypto_operation_t operation)

6.1.2 Field Documentation

6.1.2.1 te_crypto_operation_info_t __te_crypto_operation_t::info

6.1.2.2 void∗ __te_crypto_operation_t::key

6.1.2.3 void∗ __te_crypto_operation_t::iv

6.1.2.4 size_t __te_crypto_operation_t::iv_len

6.1.2.5 void∗ __te_crypto_operation_t::imp_obj

6.1.2.6 te_error_t(∗ __te_crypto_operation_t::init)(te_crypto_operation_t operation)

6.1.2.7 te_error_t(∗ __te_crypto_operation_t::update)(te_crypto_operation_t operation, const void ∗src_data, size_t src_size, void ∗dst_dat, size_t ∗dst_size)

6.1.2.8 te_error_t(∗ __te_crypto_operation_t::do_final)(te_crypto_operation_t operation, const void ∗srd_data, size_t src_size, void ∗dst_data, size_t ∗dst_size)

6.1.2.9 te_error_t(∗ __te_crypto_operation_t::handle_req)(te_crypto_operation_t operation, const void ∗src_data, size_t src_size, void ∗dst_data, size_t ∗dst_size)

6.1.2.10 void(∗ __te_crypto_operation_t::free)(te_crypto_operation_t operation)

The documentation for this struct was generated from the following file:

- ote_crypto.h

6.2 ote_closesession Struct Reference

6.2.1 Detailed Description

Closes an OTE session.

Data Fields

- uint32_t session_id
- cmnptr_t answer

6.2.2 Field Documentation

6.2.2.1 uint32_t ote_closesession::session_id

6.2.2.2 cmnptr_t ote_closesession::answer

The documentation for this struct was generated from the following file:

- ote_client.h

6.3 ote_file_close_params_t Struct Reference

Data Fields

- uint32_t handle

6.3.1 Field Documentation

6.3.1.1 uint32_t ote_file_close_params_t::handle

The documentation for this struct was generated from the following file:

- ote_client.h

6.4 ote_file_create_params_t Struct Reference

Data Fields

- char dname [OTE_MAX_DIR_NAME_LEN]
- char fname [OTE_MAX_FILE_NAME_LEN]
- uint32_t flags

6.4.1 Field Documentation

6.4.1.1 char ote_file_create_params_t::dname[OTE_MAX_DIR_NAME_LEN]

6.4.1.2 char ote_file_create_params_t::fname[OTE_MAX_FILE_NAME_LEN]

6.4.1.3 uint32_t ote_file_create_params_t::flags

The documentation for this struct was generated from the following file:

- ote_client.h

6.5 ote_file_delete_params_t Struct Reference

Data Fields

- char dname [OTE_MAX_DIR_NAME_LEN]
- char fname [OTE_MAX_FILE_NAME_LEN]

6.5.1 Field Documentation

6.5.1.1 char ote_file_delete_params_t::dname[OTE_MAX_DIR_NAME_LEN]

6.5.1.2 char ote_file_delete_params_t::fname[OTE_MAX_FILE_NAME_LEN]

The documentation for this struct was generated from the following file:

- ote_client.h

6.6 ote_file_get_size_params_t Struct Reference

Data Fields

- uint32_t handle
- uint32_t size

6.6.1 Field Documentation

6.6.1.1 uint32_t ote_file_get_size_params_t::handle

6.6.1.2 uint32_t ote_file_get_size_params_t::size

The documentation for this struct was generated from the following file:

- ote_client.h

6.7 ote_file_open_params_t Struct Reference

Data Fields

- char dname [OTE_MAX_DIR_NAME_LEN]
- char fname [OTE_MAX_FILE_NAME_LEN]
- uint32_t flags
- uint32_t handle

6.7.1 Field Documentation

6.7.1.1 char ote_file_open_params_t::dname[OTE_MAX_DIR_NAME_LEN]

6.7.1.2 char ote_file_open_params_t::fname[OTE_MAX_FILE_NAME_LEN]

6.7.1.3 uint32_t ote_file_open_params_t::flags

6.7.1.4 uint32_t ote_file_open_params_t::handle

The documentation for this struct was generated from the following file:

- ote_client.h

6.8 ote_file_read_params_t Struct Reference

Data Fields

- uint32_t handle
- uint32_t data_size
- char data [OTE_MAX_DATA_SIZE]

6.8.1 Field Documentation

6.8.1.1 uint32_t ote_file_read_params_t::handle

6.8.1.2 uint32_t ote_file_read_params_t::data_size

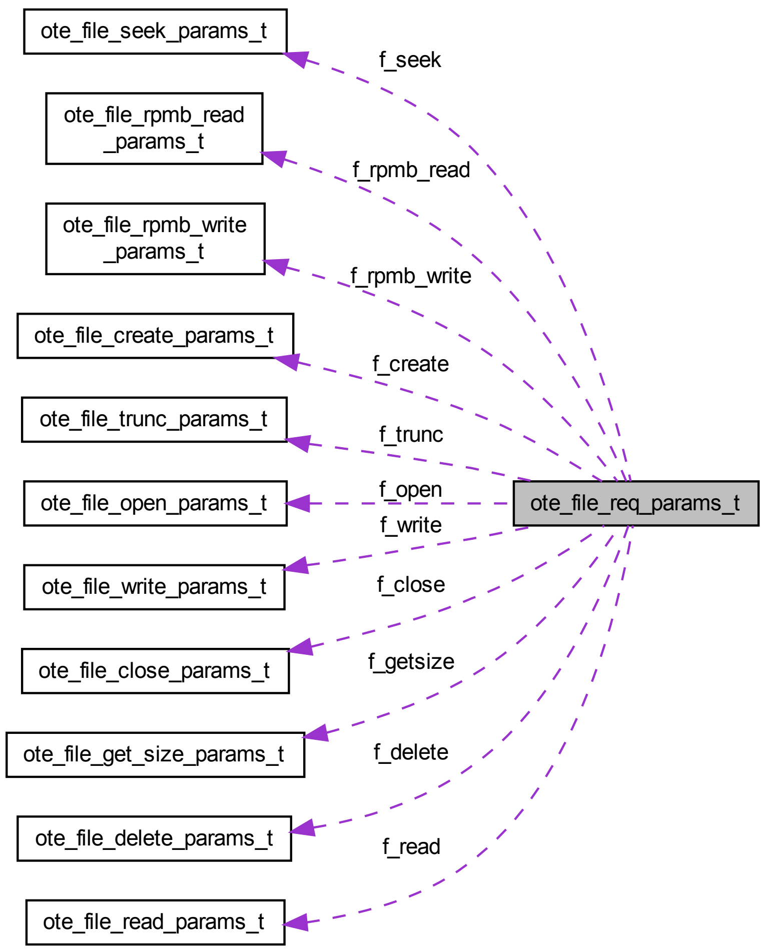
6.8.1.3 char ote_file_read_params_t::data[OTE_MAX_DATA_SIZE]

The documentation for this struct was generated from the following file:

- ote_client.h

6.9 ote_file_req_params_t Union Reference

Collaboration diagram for ote_file_req_params_t:



Data Fields

- ote_file_create_params_t f_create

- ote_file_delete_params_t f_delete
- ote_file_open_params_t f_open
- ote_file_close_params_t f_close
- ote_file_read_params_t f_read
- ote_file_write_params_t f_write
- ote_file_seek_params_t f_seek
- ote_file_trunc_params_t f_trunc
- ote_file_get_size_params_t f_getsize
- ote_file_rpmb_write_params_t f_rpmb_write
- ote_file_rpmb_read_params_t f_rpmb_read

6.9.1 Field Documentation

6.9.1.1 ote_file_create_params_t ote_file_req_params_t::f_create

6.9.1.2 ote_file_delete_params_t ote_file_req_params_t::f_delete

6.9.1.3 ote_file_open_params_t ote_file_req_params_t::f_open

6.9.1.4 ote_file_close_params_t ote_file_req_params_t::f_close

6.9.1.5 ote_file_read_params_t ote_file_req_params_t::f_read

6.9.1.6 ote_file_write_params_t ote_file_req_params_t::f_write

6.9.1.7 ote_file_seek_params_t ote_file_req_params_t::f_seek

6.9.1.8 ote_file_trunc_params_t ote_file_req_params_t::f_trunc

6.9.1.9 ote_file_get_size_params_t ote_file_req_params_t::f_getsize

6.9.1.10 ote_file_rpmb_write_params_t ote_file_req_params_t::f_rpmb_write

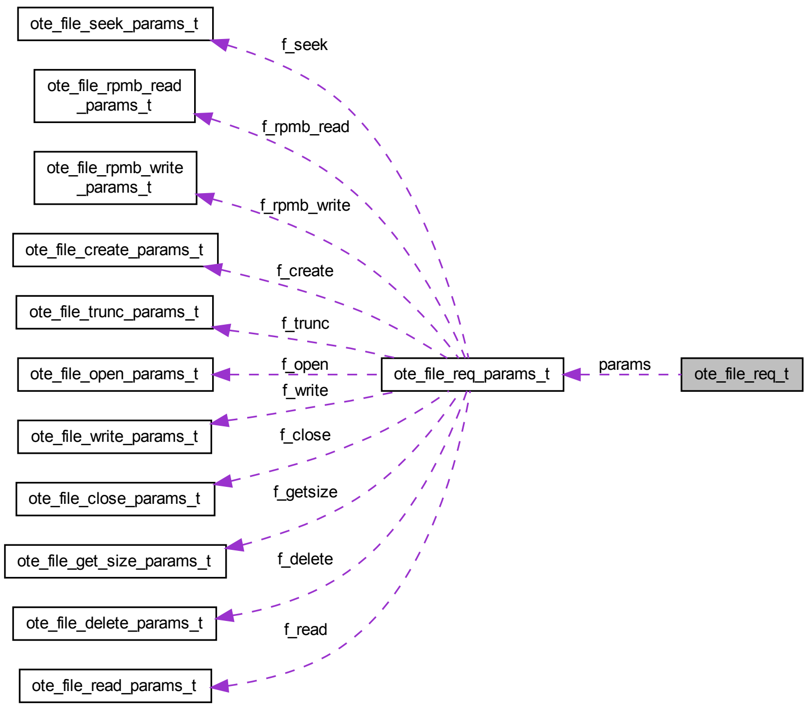
6.9.1.11 ote_file_rpmb_read_params_t ote_file_req_params_t::f_rpmb_read

The documentation for this union was generated from the following file:

- ote_client.h

6.10 ote_file_req_t Struct Reference

Collaboration diagram for ote_file_req_t:



Data Fields

- uint32_t type
- int32_t result
- uint32_t params_size
- ote_file_req_params_t params

6.10.1 Field Documentation

6.10.1.1 uint32_t ote_file_req_t::type

6.10.1.2 int32_t ote_file_req_t::result

6.10.1.3 uint32_t ote_file_req_t::params_size

6.10.1.4 ote_file_req_params_t ote_file_req_t::params

The documentation for this struct was generated from the following file:

- ote_client.h

6.11 ote_file_rpmb_read_params_t Struct Reference

Data Fields

- uint8_t req_frame [OTE_RPMB_FRAME_SIZE]
- uint8_t resp_frame [OTE_RPMB_FRAME_SIZE]

6.11.1 Field Documentation

6.11.1.1 uint8_t ote_file_rpmb_read_params_t::req_frame[OTE_RPMB_FRAME_SIZE]

6.11.1.2 uint8_t ote_file_rpmb_read_params_t::resp_frame[OTE_RPMB_FRAME_SIZE]

The documentation for this struct was generated from the following file:

- ote_client.h

6.12 ote_file_rpmb_write_params_t Struct Reference

Data Fields

- uint8_t req_frame [OTE_RPMB_FRAME_SIZE]
- uint8_t req_resp_frame [OTE_RPMB_FRAME_SIZE]
- uint8_t resp_frame [OTE_RPMB_FRAME_SIZE]

6.12.1 Field Documentation

6.12.1.1 uint8_t ote_file_rpmb_write_params_t::req_frame[OTE_RPMB_FRAME_SIZE]

6.12.1.2 uint8_t ote_file_rpmb_write_params_t::req_resp_frame[OTE_RPMB_FRAME_SIZE]

6.12.1.3 uint8_t ote_file_rpmb_write_params_t::resp_frame[OTE_RPMB_FRAME_SIZE]

The documentation for this struct was generated from the following file:

- ote_client.h

6.13 ote_file_seek_params_t Struct Reference

Data Fields

- uint32_t handle
- int32_t offset
- uint32_t whence

6.13.1 Field Documentation

6.13.1.1 uint32_t ote_file_seek_params_t::handle

6.13.1.2 int32_t ote_file_seek_params_t::offset

6.13.1.3 uint32_t ote_file_seek_params_t::whence

The documentation for this struct was generated from the following file:

- ote_client.h

6.14 ote_file_trunc_params_t Struct Reference

Data Fields

- uint32_t handle
- uint32_t length

6.14.1 Field Documentation

6.14.1.1 uint32_t ote_file_trunc_params_t::handle

6.14.1.2 uint32_t ote_file_trunc_params_t::length

The documentation for this struct was generated from the following file:

- ote_client.h

6.15 ote_file_write_params_t Struct Reference

Data Fields

- uint32_t handle
- uint32_t data_size
- char data [OTE_MAX_DATA_SIZE]

6.15.1 Field Documentation

6.15.1.1 uint32_t ote_file_write_params_t::handle

6.15.1.2 uint32_t ote_file_write_params_t::data_size

6.15.1.3 char ote_file_write_params_t::data[OTE_MAX_DATA_SIZE]

The documentation for this struct was generated from the following file:

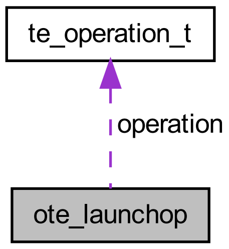
- ote_client.h

6.16 ote_launchop Struct Reference

6.16.1 Detailed Description

Launches an operation request.

Collaboration diagram for ote_launchop:



Data Fields

- uint32_t session_id
- te_operation_t operation
- cmnptr_t answer

6.16.2 Field Documentation

6.16.2.1 uint32_t ote_launchop::session_id

6.16.2.2 te_operation_t ote_launchop::operation

6.16.2.3 cmnptr_t ote_launchop::answer

The documentation for this struct was generated from the following file:

- ote_client.h

6.17 OTE_MANIFEST Struct Reference

6.17.1 Detailed Description

Holds the manifest structure.

The layout of the .ote.manifest section in the Trusted Application (TA) is the fixed fields followed by an arbitrary number of configuration options.

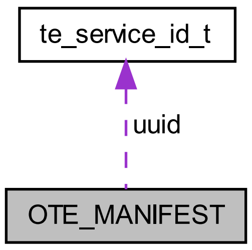
Fixed fields:

- *name* : Informative name of the task (optional).
- *private_data* : Each TA specifies how to use this field (optional).
- *uuid* : Uniquely identifies each TA in the system.

Optional 32-bit config options:

- *config_options*[]: Task configuration options set by macros in ote_config_key_t.

Collaboration diagram for OTE_MANIFEST:



Data Fields

- char name [OTE_TASK_NAME_MAX_LENGTH]
- char private_data [OTE_TASK_PRIVATE_DATA_LENGTH]
- te_service_id_t uuid
- uint32_t config_options [ ]

6.17.2 Field Documentation

6.17.2.1 char OTE_MANIFEST::name[OTE_TASK_NAME_MAX_LENGTH]

6.17.2.2 char OTE_MANIFEST::private_data[OTE_TASK_PRIVATE_DATA_LENGTH]

6.17.2.3 te_service_id_t OTE_MANIFEST::uuid

6.17.2.4 uint32_t OTE_MANIFEST::config_options[ ]

The documentation for this struct was generated from the following file:

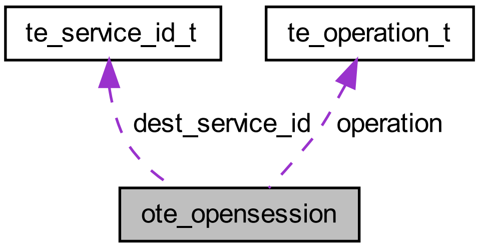
- ote_manifest.h

6.18 ote_opensession Struct Reference

6.18.1 Detailed Description

Opens an open trusted environment (OTE) session.

Collaboration diagram for ote_opensession:



Data Fields

- te_service_id_t dest_service_id
- te_operation_t operation
- cmnptr_t answer

6.18.2 Field Documentation

6.18.2.1 te_service_id_t ote_opensession::dest_service_id

6.18.2.2 te_operation_t ote_opensession::operation

6.18.2.3 cmnptr_t ote_opensession::answer

The documentation for this struct was generated from the following file:

- ote_client.h

6.19 ote_ss_op_t Struct Reference

Data Fields

- uint32_t req_size
- uint8_t data [SS_OP_MAX_DATA_SIZE]

6.19.1 Field Documentation

6.19.1.1 uint32_t ote_ss_op_t::req_size

6.19.1.2 uint8_t ote_ss_op_t::data[SS_OP_MAX_DATA_SIZE]

The documentation for this struct was generated from the following file:

- ote_client.h

6.20 ta_event_args_t Struct Reference

Data Fields

- enum ta_event_id event_id

6.20.1 Field Documentation

6.20.1.1 enum ta_event_id ta_event_args_t::event_id

The documentation for this struct was generated from the following file:

- ote_service.h

6.21 te_answer Struct Reference

Data Fields

- uint32_t result
- uint32_t session_id
- uint32_t result_origin

6.21.1 Field Documentation

6.21.1.1 uint32_t te_answer::result

6.21.1.2 uint32_t te_answer::session_id

6.21.1.3 uint32_t te_answer::result_origin

The documentation for this struct was generated from the following file:

- ote_client.h

6.22 te_app_block_args_t Struct Reference

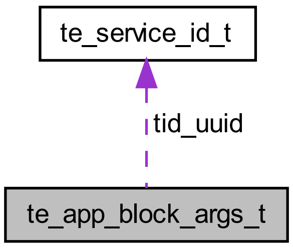
6.22.1 Detailed Description

Holds information that is passed to the ioctl handler for the OTE_TASK_OP_BLOCK operation.

See Also

te_task_request_args_t

Collaboration diagram for te_app_block_args_t:



Data Fields

- te_app_id_t tid_type
- union {
  uint32_t tid_index
  te_service_id_t tid_uuid
  };

6.22.2 Field Documentation

6.22.2.1 te_app_id_t te_app_block_args_t::tid_type

6.22.2.2 uint32_t te_app_block_args_t::tid_index

6.22.2.3 te_service_id_t te_app_block_args_t::tid_uuid

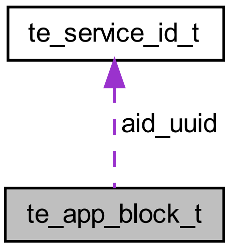
6.22.2.4 union { ... }

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.23 te_app_block_t Struct Reference

Collaboration diagram for te_app_block_t:



Data Fields

- te_app_id_t aid_type
- union {
  uint32_t aid_index
  te_service_id_t aid_uuid
  };

6.23.1 Field Documentation

6.23.1.1 te_app_id_t te_app_block_t::aid_type

6.23.1.2 uint32_t te_app_block_t::aid_index

6.23.1.3 te_service_id_t te_app_block_t::aid_uuid

6.23.1.4 union { ... }

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.24 te_app_list_args_t Struct Reference

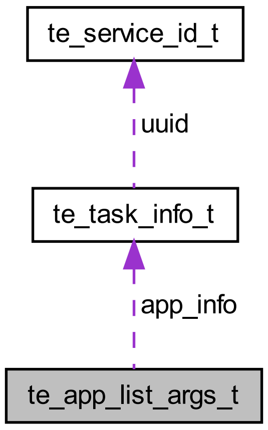
6.24.1 Detailed Description

Holds information that is passed to the ioctl handler for OTE_TASK_OP_LIST operations.

See Also

te_task_request_args_t

Collaboration diagram for te_app_list_args_t:



Data Fields

- uint32_t app_index
- uint32_t app_type
- uint32_t app_state
- te_task_info_t app_info

6.24.2 Field Documentation

6.24.2.1 uint32_t te_app_list_args_t::app_index

6.24.2.2 uint32_t te_app_list_args_t::app_type

6.24.2.3 uint32_t te_app_list_args_t::app_state

6.24.2.4 te_task_info_t te_app_list_args_t::app_info

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.25 te_app_load_memory_request_args_t Struct Reference

6.25.1 Detailed Description

Holds information that is passed to the ioctl handler for the OTE_TASK_OP_MEMORY_REQUEST operation.

See Also

te_task_request_args_t

Data Fields

- uint32_t app_handle
- uintptr_t app_addr
- uint32_t app_size

6.25.2 Field Documentation

6.25.2.1 uint32_t te_app_load_memory_request_args_t::app_handle

6.25.2.2 uintptr_t te_app_load_memory_request_args_t::app_addr

6.25.2.3 uint32_t te_app_load_memory_request_args_t::app_size

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.26 te_app_prepare_args_t Struct Reference

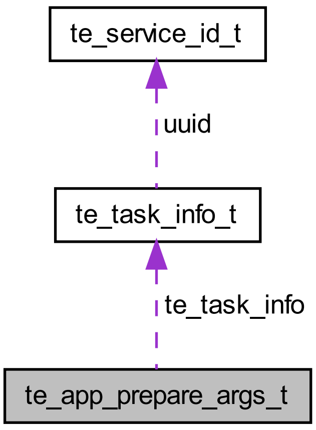
6.26.1 Detailed Description

Holds information passed to the ioctl handler with the OTE_TASK_OP_PREPARE operation.

See Also

te_task_request_args_t

Collaboration diagram for te_app_prepare_args_t:



Data Fields

- uint32_t app_handle
- te_task_info_t te_task_info

6.26.2 Field Documentation

6.26.2.1 uint32_t te_app_prepare_args_t::app_handle

6.26.2.2 te_task_info_t te_app_prepare_args_t::te_task_info

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.27 te_app_start_args_t Struct Reference

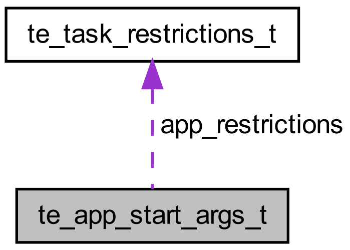
6.27.1 Detailed Description

Holds information that is passed to the ioctl handler for OTE_TASK_OP_START operations.

See Also

te_task_request_args_t

Collaboration diagram for te_app_start_args_t:



Data Fields

- uint32_t app_handle
- install_type_t app_install_type
- te_task_restrictions_t app_restrictions

6.27.2 Field Documentation

6.27.2.1 uint32_t te_app_start_args_t::app_handle

6.27.2.2 install_type_t te_app_start_args_t::app_install_type

6.27.2.3 te_task_restrictions_t te_app_start_args_t::app_restrictions

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.28 te_app_unload_args_t Struct Reference

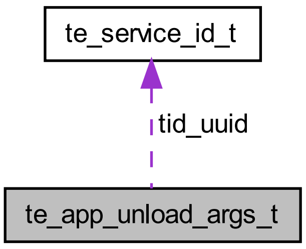
6.28.1 Detailed Description

Holds information that is passed to the ioctl handler for the OTE_TASK_OP_UNLOAD operation.

See Also

te_task_request_args_t

Collaboration diagram for te_app_unload_args_t:



Data Fields

- te_app_id_t tid_type
- union {
  uint32_t tid_index
  te_service_id_t tid_uuid
  };

6.28.2 Field Documentation

6.28.2.1 te_app_id_t te_app_unload_args_t::tid_type

6.28.2.2 uint32_t te_app_unload_args_t::tid_index

6.28.2.3 te_service_id_t te_app_unload_args_t::tid_uuid

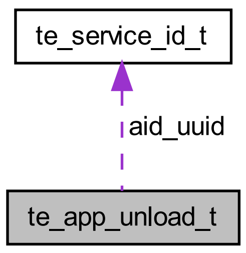
6.28.2.4 union { ... }

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.29 te_app_unload_t Struct Reference

Collaboration diagram for te_app_unload_t:



Data Fields

- te_app_id_t aid_type
- union {
    uint32_t aid_index
    te_service_id_t aid_uuid
  };

6.29.1 Field Documentation

6.29.1.1 te_app_id_t te_app_unload_t::aid_type

6.29.1.2 uint32_t te_app_unload_t::aid_index

6.29.1.3 te_service_id_t te_app_unload_t::aid_uuid

6.29.1.4 union { ... }

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.30 te_attribute_t Struct Reference

6.30.1 Detailed Description

Defines attribute object internals. Attributes can be integer or reference type.

Data Fields

- te_attribute_id_t id
- union {
  struct {
  void ∗ buffer
  size_t size
  } ref
  struct {
  uint32_t a
  uint32_t b
  } value
  } content

6.30.2 Field Documentation

6.30.2.1 te_attribute_id_t te_attribute_t::id

6.30.2.2 void∗ te_attribute_t::buffer

6.30.2.3 size_t te_attribute_t::size

6.30.2.4 struct { ... } te_attribute_t::ref

6.30.2.5 uint32_t te_attribute_t::a

6.30.2.6 uint32_t te_attribute_t::b

6.30.2.7 struct { ... } te_attribute_t::value

6.30.2.8 union { ... } te_attribute_t::content

The documentation for this struct was generated from the following file:

- ote_attrs.h

6.31 te_cache_maint_args_t Struct Reference

6.31.1 Detailed Description

Holds an op code and data used to for cache maintenance.

Used with the OTE_IOCTL_CACHE_MAINT operation.

Data Fields

- void ∗ vaddr

  Holds a pointer to the virtual address.

- uint32_t length

    Holds the length of the address space.

- uint32_t op

    Holds the operation; see te_ext_nv_cache_maint_op_t.

6.31.2 Field Documentation

6.31.2.1 void∗ te_cache_maint_args_t::vaddr

Holds a pointer to the virtual address.

6.31.2.2 uint32_t te_cache_maint_args_t::length

Holds the length of the address space.

6.31.2.3 uint32_t te_cache_maint_args_t::op

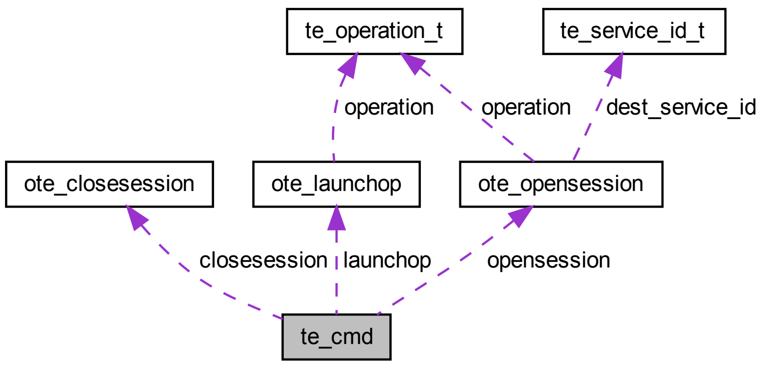
Holds the operation; see te_ext_nv_cache_maint_op_t.

The documentation for this struct was generated from the following file:

- ote_ext_nv.h

6.32 te_cmd Union Reference

Collaboration diagram for te_cmd:



Data Fields

- struct ote_opensession opensession
- struct ote_closesession closesession
- struct ote_launchop launchop

6.32.1 Field Documentation

6.32.1.1 struct ote_opensession te_cmd::opensession

6.32.1.2 struct ote_closesession te_cmd::closesession

6.32.1.3 struct ote_launchop te_cmd::launchop

The documentation for this union was generated from the following file:

- ote_client.h

6.33 te_crypto_operation_info_t Struct Reference

6.33.1 Detailed Description

Holds a crypto operation info object.

Data Fields

- uint32_t algorithm
- uint32_t operation_class
- uint32_t mode
- uint32_t digest_length
- uint32_t key_size
- uint32_t required_key_usage
- uint32_t handle_state

6.33.2 Field Documentation

6.33.2.1 uint32_t te_crypto_operation_info_t::algorithm

6.33.2.2 uint32_t te_crypto_operation_info_t::operation_class

6.33.2.3 uint32_t te_crypto_operation_info_t::mode

6.33.2.4 uint32_t te_crypto_operation_info_t::digest_length

6.33.2.5 uint32_t te_crypto_operation_info_t::key_size

6.33.2.6 uint32_t te_crypto_operation_info_t::required_key_usage

6.33.2.7 uint32_t te_crypto_operation_info_t::handle_state

The documentation for this struct was generated from the following file:

- ote_crypto.h

6.34 te_crypto_rsa_key_t Struct Reference

6.34.1 Detailed Description

Holds internal data for RSA keys.

Data Fields

- uint8_t ∗ public_mod
- int public_mod_len
- uint8_t ∗ public_expo
- int public_expo_len
- uint8_t ∗ private_expo
- int private_expo_len
- uint8_t ∗ prime1
- int prime1_len
- uint8_t ∗ prime2
- int prime2_len
- uint8_t ∗ expo1
- int expo1_len
- uint8_t ∗ expo2
- int expo2_len
- uint8_t ∗ coeff
- int coeff_len

6.34.2 Field Documentation

6.34.2.1 uint8_t∗ te_crypto_rsa_key_t::public_mod

Holds public modulus.

6.34.2.2 int te_crypto_rsa_key_t::public_mod_len

Holds public modulus length in bytes.

6.34.2.3 uint8_t∗ te_crypto_rsa_key_t::public_expo

Holds public exponent.

6.34.2.4 int te_crypto_rsa_key_t::public_expo_len

Holds public exponent length in bytes.

6.34.2.5 uint8_t∗ te_crypto_rsa_key_t::private_expo

Holds private exponent.

6.34.2.6 int te_crypto_rsa_key_t::private_expo_len

Holds private exponent length in bytes.

6.34.2.7 uint8_t∗ te_crypto_rsa_key_t::prime1

Holds secret prime factor.

6.34.2.8 int te_crypto_rsa_key_t::prime1_len

Holds *prime1* length in bytes.

6.34.2.9 uint8_t∗ te_crypto_rsa_key_t::prime2

Holds secret prime factor.

6.34.2.10 int te_crypto_rsa_key_t::prime2_len

Holds *prime2* length in bytes.

6.34.2.11 uint8_t∗ te_crypto_rsa_key_t::expo1

Holds d mod (p-1).

6.34.2.12 int te_crypto_rsa_key_t::expo1_len

Holds *expo1* length in bytes.

6.34.2.13 uint8_t∗ te_crypto_rsa_key_t::expo2

Holds d mod (q-1).

6.34.2.14 int te_crypto_rsa_key_t::expo2_len

Holds *expo2* length in bytes.

6.34.2.15 uint8_t∗ te_crypto_rsa_key_t::coeff

Holds q^-1 mod p.

6.34.2.16 int te_crypto_rsa_key_t::coeff_len

Holds the coefficient length in bytes.

The documentation for this struct was generated from the following file:

- ote_crypto.h

6.35 te_device_unique_id Struct Reference

6.35.1 Detailed Description

Holds the device unique ID.

Data Fields

- uint8_t id [DEVICE_UID_SIZE_BYTES]

6.35.2 Field Documentation

6.35.2.1 uint8_t te_device_unique_id::id[DEVICE_UID_SIZE_BYTES]

The documentation for this struct was generated from the following file:

- ote_service.h

6.36 te_entry_point_message_t Struct Reference

Data Fields

- uint32_t type
- te_error_t result
- cmnptr_t context
- uint32_t command_id
- cmnptr_t params
- uint32_t params_size

6.36.1 Field Documentation

6.36.1.1 uint32_t te_entry_point_message_t::type

6.36.1.2 te_error_t te_entry_point_message_t::result

6.36.1.3 cmnptr_t te_entry_point_message_t::context

6.36.1.4 uint32_t te_entry_point_message_t::command_id

6.36.1.5 cmnptr_t te_entry_point_message_t::params

6.36.1.6 uint32_t te_entry_point_message_t::params_size

The documentation for this struct was generated from the following file:

- ote_service.h

6.37 te_get_pending_map_args_t Struct Reference

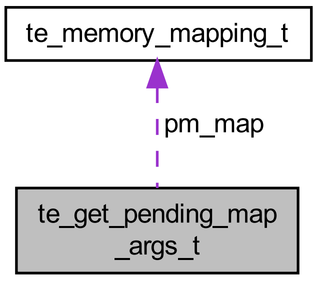
6.37.1 Detailed Description

Holds information that is passed to the ioctl handler for the OTE_TASK_OP_PENDING_MAPPING operation.

See Also

te_task_request_args_t

Collaboration diagram for te_get_pending_map_args_t:



Data Fields

- uint32_t pm_handle
- te_memory_mapping_t pm_map

6.37.2 Field Documentation

6.37.2.1 uint32_t te_get_pending_map_args_t::pm_handle

6.37.2.2 te_memory_mapping_t te_get_pending_map_args_t::pm_map

The documentation for this struct was generated from the following file:

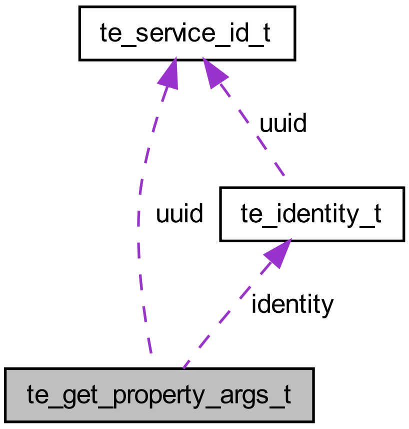
- ote_task_load.h

6.38 te_get_property_args_t Struct Reference

6.38.1 Detailed Description

Holds data about the TA client.

Collaboration diagram for te_get_property_args_t:



Data Fields

- te_property_type_t prop

  Holds the TE_PROPERTY_∗ value.
- uint32_t data_type

  Holds the data type of property.
- union {
  te_service_id_t uuid
  te_identity_t identity
  } value

  Holds the return value.
- size_t value_size

  Holds the size of return value.
- te_error_t result

6.38.2 Field Documentation

6.38.2.1 te_property_type_t te_get_property_args_t::prop

Holds the TE_PROPERTY_∗ value.

6.38.2.2 uint32_t te_get_property_args_t::data_type

Holds the data type of property.

6.38.2.3 te_service_id_t te_get_property_args_t::uuid

6.38.2.4 te_identity_t te_get_property_args_t::identity

6.38.2.5 union { ... } te_get_property_args_t::value

Holds the return value.

6.38.2.6 size_t te_get_property_args_t::value_size

Holds the size of return value.

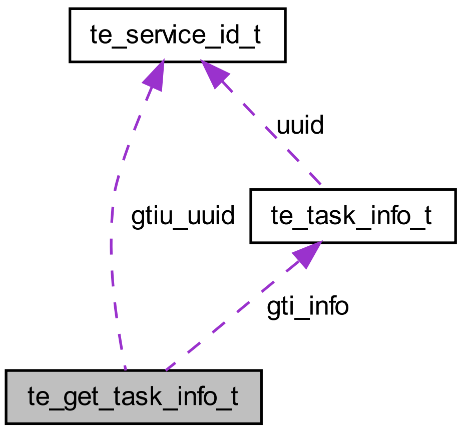
6.38.2.7 te_error_t te_get_property_args_t::result

The documentation for this struct was generated from the following file:

- ote_service.h

6.39 te_get_task_info_t Struct Reference

Collaboration diagram for te_get_task_info_t:



Data Fields

- te_get_info_type_t gti_request_type
- union {
    uint32_t gtiu_index
    te_service_id_t gtiu_uuid
  };
- uint32_t gti_state
- uint32_t gti_type
- te_task_info_t gti_info

6.39.1 Field Documentation

6.39.1.1 te_get_info_type_t te_get_task_info_t::gti_request_type

6.39.1.2 uint32_t te_get_task_info_t::gtiu_index

6.39.1.3 te_service_id_t te_get_task_info_t::gtiu_uuid

6.39.1.4 union { ... }

6.39.1.5 uint32_t te_get_task_info_t::gti_state

6.39.1.6 uint32_t te_get_task_info_t::gti_type

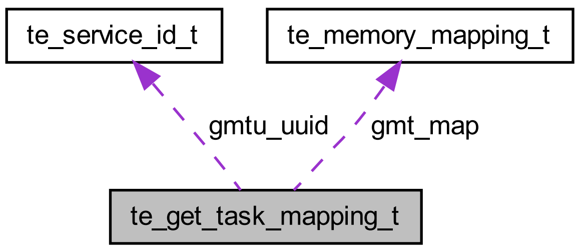
6.39.1.7 te_task_info_t te_get_task_info_t::gti_info

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.40 te_get_task_mapping_t Struct Reference

Collaboration diagram for te_get_task_mapping_t:



Data Fields

- te_get_info_type_t gmt_request_type
- union {
  uint32_t gmtu_index
  te_service_id_t gmtu_uuid
  };
- te_memory_mapping_t gmt_map

6.40.1 Field Documentation

6.40.1.1 te_get_info_type_t te_get_task_mapping_t::gmt_request_type

6.40.1.2 uint32_t te_get_task_mapping_t::gmtu_index

6.40.1.3 te_service_id_t te_get_task_mapping_t::gmtu_uuid

6.40.1.4 union { ... }

6.40.1.5 te_memory_mapping_t te_get_task_mapping_t::gmt_map

The documentation for this struct was generated from the following file:

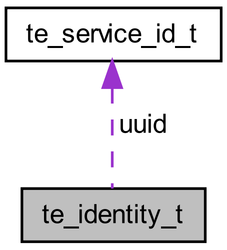
- ote_task_load.h

6.41 te_identity_t Struct Reference

6.41.1 Detailed Description

Holds the identity of a client/caller.

Collaboration diagram for te_identity_t:



Data Fields

- uint32_t login
- te_service_id_t uuid

6.41.2 Field Documentation

6.41.2.1 uint32_t te_identity_t::login

6.41.2.2 te_service_id_t te_identity_t::uuid

The documentation for this struct was generated from the following file:

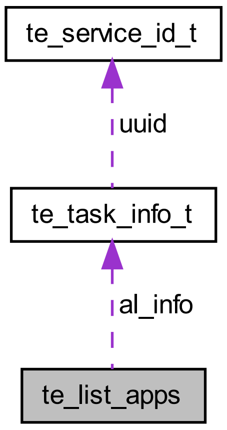
- ote_service.h

6.42 te_list_apps Struct Reference

6.42.1 Detailed Description

Holds list command returned information.

Collaboration diagram for te_list_apps:



Data Fields

- uint32_t al_index
- uint32_t al_type
- uint32_t al_state
- te_task_info_t al_info

6.42.2 Field Documentation

6.42.2.1 uint32_t te_list_apps::al_index

6.42.2.2 uint32_t te_list_apps::al_type

6.42.2.3 uint32_t te_list_apps::al_state

6.42.2.4 te_task_info_t te_list_apps::al_info

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.43 te_map_mem_addr_args_t Struct Reference

6.43.1 Detailed Description

Holds a pointer to the map memory for a specific OTE_CONFIG_MAP_MEM ID value.

Used with the OTE_IOCTL_GET_MAP_MEM_ADDR operation.

Data Fields

- uint32_t id

  Holds the OTE_CONFIG_MAP_MEM ID value.
- void ∗ addr

  Holds a pointer to the corresponding address of mapping.

6.43.2 Field Documentation

6.43.2.1 uint32_t te_map_mem_addr_args_t::id

Holds the OTE_CONFIG_MAP_MEM ID value.

6.43.2.2 void∗ te_map_mem_addr_args_t::addr

Holds a pointer to the corresponding address of mapping.

The documentation for this struct was generated from the following file:

- ote_ext_nv.h

6.44 te_memory_mapping_t Struct Reference

Data Fields

- uint32_t map_index
- uint32_t map_id
- uint32_t map_offset
- uint32_t map_size

6.44.1 Field Documentation

6.44.1.1 uint32_t te_memory_mapping_t::map_index

6.44.1.2 uint32_t te_memory_mapping_t::map_id

6.44.1.3 uint32_t te_memory_mapping_t::map_offset

6.44.1.4 uint32_t te_memory_mapping_t::map_size

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.45 te_oper_param_t Struct Reference

6.45.1 Detailed Description

Holds the operation object parameters.

Data Fields

- uint32_t index
- te_oper_param_type_t type
- union {
    struct {
        uint32_t val
    } Int
    struct {
        cmnptr_t base
        uint32_t len
    } Mem
  } u
- cmnptr_t next

6.45.2 Field Documentation

6.45.2.1 uint32_t te_oper_param_t::index

6.45.2.2 te_oper_param_type_t te_oper_param_t::type

6.45.2.3 uint32_t te_oper_param_t::val

6.45.2.4 struct { ... } te_oper_param_t::Int

6.45.2.5 cmnptr_t te_oper_param_t::base

6.45.2.6 uint32_t te_oper_param_t::len

6.45.2.7 struct { ... } te_oper_param_t::Mem

6.45.2.8 union { ... } te_oper_param_t::u

6.45.2.9 cmnptr_t te_oper_param_t::next

The documentation for this struct was generated from the following file:

- ote_common.h

6.46 te_operation_t Struct Reference

6.46.1 Detailed Description

Holds operation object information that is to be delivered to the TLK Secure Service.

Data Fields

- uint32_t command
- te_error_t status
- cmnptr_t list_head

  Holds pointers to the head/tail of the list of param_t nodes.
- cmnptr_t list_tail

- uint32_t list_count
- uint32_t interface_side

6.46.2 Field Documentation

6.46.2.1 uint32_t te_operation_t::command

6.46.2.2 te_error_t te_operation_t::status

6.46.2.3 cmnptr_t te_operation_t::list_head

Holds pointers to the head/tail of the list of *param_t* nodes.

6.46.2.4 cmnptr_t te_operation_t::list_tail

6.46.2.5 uint32_t te_operation_t::list_count

6.46.2.6 uint32_t te_operation_t::interface_side

The documentation for this struct was generated from the following file:

- ote_common.h

6.47 te_panic_args_t Struct Reference

Data Fields

- char msg [OTE_PANIC_MSG_MAX_SIZE+1]

6.47.1 Field Documentation

6.47.1.1 char te_panic_args_t::msg[OTE_PANIC_MSG_MAX_SIZE+1]

The documentation for this struct was generated from the following file:

- ote_service.h

6.48 te_request_t Struct Reference

6.48.1 Detailed Description

Holds the layout of the `te_oper_param_t` structures which must match the layout sent in by the non-secure (NS) world via the TrustZone Secure Monitor Call (TZ SMC) path.

Data Fields

- uint32_t type
- uint32_t session_id
- uint32_t command_id
- cmnptr_t params

- uint32_t params_size
- uint32_t dest_uuid [4]
- uint32_t result
- uint32_t result_origin

6.48.2 Field Documentation

6.48.2.1 uint32_t te_request_t::type

6.48.2.2 uint32_t te_request_t::session_id

6.48.2.3 uint32_t te_request_t::command_id

6.48.2.4 cmnptr_t te_request_t::params

6.48.2.5 uint32_t te_request_t::params_size

6.48.2.6 uint32_t te_request_t::dest_uuid[4]

6.48.2.7 uint32_t te_request_t::result

6.48.2.8 uint32_t te_request_t::result_origin

The documentation for this struct was generated from the following file:

- ote_service.h

6.49 te_service_id_t Struct Reference

6.49.1 Detailed Description

Defines a unique 16-byte ID for each TLK service.

Data Fields

- uint32_t time_low
- uint16_t time_mid
- uint16_t time_hi_and_version
- uint8_t clock_seq_and_node [8]

6.49.2 Field Documentation

6.49.2.1 uint32_t te_service_id_t::time_low

6.49.2.2 uint16_t te_service_id_t::time_mid

6.49.2.3 uint16_t te_service_id_t::time_hi_and_version

6.49.2.4 uint8_t te_service_id_t::clock_seq_and_node[8]

The documentation for this struct was generated from the following file:

- ote_common.h

6.50 te_session_t Union Reference

6.50.1 Detailed Description

Holds session information.

Data Fields

- struct {
  uint32_t session_id
  uint32_t context_id
  te_result_origin_t result_origin
  } client
- struct {
  uint32_t session_id
  te_result_origin_t result_origin
  } service

6.50.2 Field Documentation

6.50.2.1 uint32_t te_session_t::session_id

6.50.2.2 uint32_t te_session_t::context_id

6.50.2.3 te_result_origin_t te_session_t::result_origin

6.50.2.4 struct { ... } te_session_t::client

6.50.2.5 struct { ... } te_session_t::service

The documentation for this union was generated from the following file:

- ote_common.h

6.51 te_system_info_args_t Struct Reference

Data Fields

- uint32_t si_type

6.51.1 Field Documentation

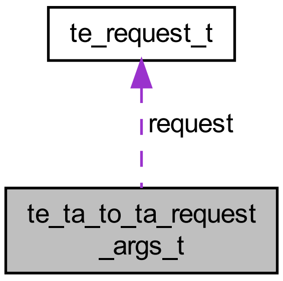
6.51.1.1 uint32_t te_system_info_args_t::si_type

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.52 te_ta_to_ta_request_args_t Struct Reference

Collaboration diagram for te_ta_to_ta_request_args_t:



Data Fields

- te_request_t request

6.52.1 Field Documentation

6.52.1.1 te_request_t te_ta_to_ta_request_args_t::request

The documentation for this struct was generated from the following file:

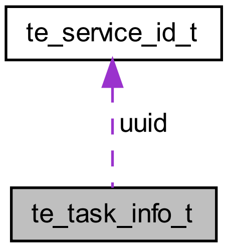
- ote_service.h

6.53 te_task_info_t Struct Reference

6.53.1 Detailed Description

This is compatible with task_info_t, field values extracted from manifest record

Collaboration diagram for te_task_info_t:



Data Fields

- te_service_id_t uuid
- u_int manifest_exists
- u_int multi_instance
- u_int min_stack_size
- u_int min_heap_size
- u_int map_io_mem_cnt
- u_int restrict_access
- u_int authorizations
- u_int initial_state
- char task_name [OTE_TASK_NAME_MAX_LENGTH]
- unsigned char task_private_data [OTE_TASK_PRIVATE_DATA_LENGTH]

6.53.2 Field Documentation

6.53.2.1 te_service_id_t te_task_info_t::uuid

6.53.2.2 u_int te_task_info_t::manifest_exists

6.53.2.3 u_int te_task_info_t::multi_instance

6.53.2.4 u_int te_task_info_t::min_stack_size

6.53.2.5 u_int te_task_info_t::min_heap_size

6.53.2.6 u_int te_task_info_t::map_io_mem_cnt

6.53.2.7 u_int te_task_info_t::restrict_access

6.53.2.8 u_int te_task_info_t::authorizations

6.53.2.9 u_int te_task_info_t::initial_state

6.53.2.10 char te_task_info_t::task_name[OTE_TASK_NAME_MAX_LENGTH]

6.53.2.11 unsigned char te_task_info_t::task_private_data[OTE_TASK_PRIVATE_DATA_LENGTH]

The documentation for this struct was generated from the following file:

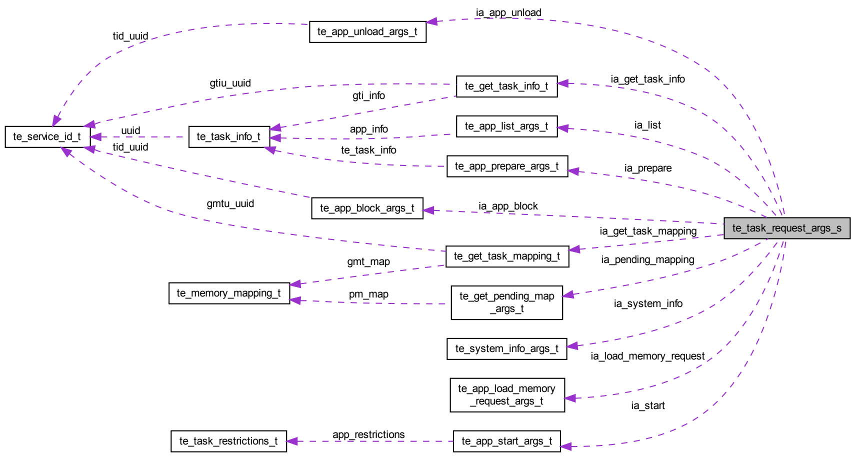
- ote_task_load.h

6.54 te_task_request_args_s Struct Reference

6.54.1 Detailed Description

Holds the task-loading ioctl argument used with OTE_IOCTL_TASK_REQUEST commands. The *te_task_request-_args_t::ia_opcode* field specifies the operation, which in turn indicates which argument from the union fields applies. For example, if *te_task_request_args_t::ia_opcode* is OTE_TASK_OP_MEMORY_REQUEST, then *te_task-_request_args_t::ia_load_memory_request* provides the memory request.

Collaboration diagram for te_task_request_args_s:



Data Fields

- te_task_opcode_t ia_opcode
- union {
  te_app_load_memory_request_args_t ia_load_memory_request
  te_app_prepare_args_t ia_prepare
  te_get_pending_map_args_t ia_pending_mapping
  te_app_start_args_t ia_start
  te_app_list_args_t ia_list
  te_get_task_info_t ia_get_task_info
  te_get_task_mapping_t ia_get_task_mapping
  te_app_unload_args_t ia_app_unload
  te_app_block_args_t ia_app_block
  te_system_info_args_t ia_system_info
  };

6.54.2 Field Documentation

6.54.2.1 te_task_opcode_t te_task_request_args_s::ia_opcode

6.54.2.2 te_app_load_memory_request_args_t te_task_request_args_s::ia_load_memory_request

6.54.2.3 te_app_prepare_args_t te_task_request_args_s::ia_prepare

6.54.2.4 te_get_pending_map_args_t te_task_request_args_s::ia_pending_mapping

6.54.2.5 te_app_start_args_t te_task_request_args_s::ia_start

6.54.2.6 te_app_list_args_t te_task_request_args_s::ia_list

6.54.2.7 te_get_task_info_t te_task_request_args_s::ia_get_task_info

6.54.2.8 te_get_task_mapping_t te_task_request_args_s::ia_get_task_mapping

6.54.2.9 te_app_unload_args_t te_task_request_args_s::ia_app_unload

6.54.2.10 te_app_block_args_t te_task_request_args_s::ia_app_block

6.54.2.11 te_system_info_args_t te_task_request_args_s::ia_system_info

6.54.2.12 union { ... }

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.55 te_task_restrictions_t Struct Reference

6.55.1 Detailed Description

Holds task restrictions requested by the installer (manifest overrides).

Note

This does not currently allow to "override" all manifest fields, e.g. the UUID which means that tasks without manifests can not be loaded.

In general: only non-security critical field value overrides can currently be requested (more/all fields can be added if required).

If the task manifest is flagged immutable, TLK will not modify any manifest values by the override mechanism. In this case the installer may choose to reject immutable tasks if overrides are mandatory (any override values will be ignored in this case). For immutable tasks the APP_RESTRICTIONS parameter must be set to NULL.

This record is compatible with task_restrictions_t

Data Fields

- uint32_t min_stack_size
- uint32_t min_heap_size
- char task_name [OTE_TASK_NAME_MAX_LENGTH]

6.55.2 Field Documentation

6.55.2.1 uint32_t te_task_restrictions_t::min_stack_size

6.55.2.2 uint32_t te_task_restrictions_t::min_heap_size

6.55.2.3 char te_task_restrictions_t::task_name[OTE_TASK_NAME_MAX_LENGTH]

The documentation for this struct was generated from the following file:

- ote_task_load.h

6.56 te_v_to_p_args_t Struct Reference

6.56.1 Detailed Description

Holds a pointer to the physical address for a specific virtual address.

Used with the OTE_IOCTL_V_TO_P operation.

Data Fields

- void ∗ vaddr

  Holds a pointer to the virtual address.
- uint64_t paddr

  Holds a pointer to the corresponding physical address.

6.56.2 Field Documentation

6.56.2.1 void∗ te_v_to_p_args_t::vaddr

Holds a pointer to the virtual address.

6.56.2.2 uint64_t te_v_to_p_args_t::paddr

Holds a pointer to the corresponding physical address.

The documentation for this struct was generated from the following file:

- ote_ext_nv.h

Chapter 7

File Documentation

7.1 nvtlkoss.txt File Reference

7.2 ote_attrs.h File Reference

7.2.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Service Attributes Description: Declares the service attributes in the TLK.

Data Structures

- struct te_attribute_t

Macros

- #define OTE_ATTR_VAL 1 $<<$ 29
- #define OTE_ATTR_PUB 1 $<<$ 28

Enumerations

- enum te_attribute_id_t {
  OTE_ATTR_SECRET_VALUE = 0xC0000000,
  OTE_ATTR_RSA_MODULES = 0xD0000130,
  OTE_ATTR_RSA_PUBLIC_EXPONENT = 0xD0000230,
  OTE_ATTR_RSA_PRIVATE_EXPONENT = 0xC0000330,
  OTE_ATTR_RSA_PRIME1 = 0xC0000430,
  OTE_ATTR_RSA_PRIME2 = 0xC0000530,
  OTE_ATTR_RSA_EXPONENT1 = 0xC0000630,
  OTE_ATTR_RSA_EXPONENT2 = 0xC0000730,
  OTE_ATTR_RSA_COEFFICIENT = 0xC0000830,
  OTE_ATTR_DSA_PRIME = 0xD0001031,
  OTE_ATTR_DSA_SUBPRIME = 0xD0001131,
  OTE_ATTR_DSA_BASE = 0xD0001231,
  OTE_ATTR_DSA_PUBLIC_VALUE = 0xD0000131,
  OTE_ATTR_DSA_PRIVATE_VALUE = 0xD0000231,
  OTE_ATTR_DH_PRIME = 0xD0001032,
  OTE_ATTR_DH_SUBPRIME = 0xD0001132,
  OTE_ATTR_DH_BASE = 0xD0001232,
  OTE_ATTR_DH_X_BITS = 0xF0001332,
  OTE_ATTR_DH_PUBLIC_VALUE = 0xD0000132,
  OTE_ATTR_DH_PRIVATE_VALUE = 0xC0000232,
  OTE_ATTR_RSA_OAEP_LABEL = 0xD0000930,
  OTE_ATTR_RSA_PSS_SALT_LENGTH = 0xF0000A30 }

Functions

- te_error_t te_set_mem_attribute (te_attribute_t ∗attr, te_attribute_id_t id, void ∗buffer, uint32_t size)
- te_error_t te_get_mem_attribute_buffer (te_attribute_t ∗attr, void ∗∗ret)
- te_error_t te_get_mem_attribute_size (te_attribute_t ∗attr, size_t ∗ret)
- void te_copy_mem_attribute (void ∗buffer, te_attribute_t ∗key)
- te_error_t te_set_int_attribute (te_attribute_t ∗attr, te_attribute_id_t id, uint32_t a, uint32_t b)
- void te_free_internal_attribute (te_attribute_t ∗attr)
- te_error_t te_copy_attribute (te_attribute_t ∗dst, te_attribute_t ∗src)

7.3 ote_client.h File Reference

7.3.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Client Communications Description: Declares the TLK client interface.

Data Structures

- struct te_answer
- struct ote_opensession

  Opens an open trusted environment (OTE) session.
- struct ote_closesession

  Closes an OTE session.
- struct ote_launchop

  Launches an operation request.
- union te_cmd
- struct ote_file_create_params_t

- struct ote_file_delete_params_t
- struct ote_file_open_params_t
- struct ote_file_close_params_t
- struct ote_file_write_params_t
- struct ote_file_read_params_t
- struct ote_file_seek_params_t
- struct ote_file_trunc_params_t
- struct ote_file_get_size_params_t
- struct ote_file_rpmb_write_params_t
- struct ote_file_rpmb_read_params_t
- union ote_file_req_params_t
- struct ote_file_req_t
- struct ote_ss_op_t

Macros

- #define TLK_DEVICE_BASE_NAME "tlk_device"
- #define TE_IOCTL_MAGIC_NUMBER ('t')
- #define TE_IOCTL_OPEN_CLIENT_SESSION _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x10, union te_cmd)
- #define TE_IOCTL_CLOSE_CLIENT_SESSION _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x11, union te_-cmd)
- #define TE_IOCTL_LAUNCH_OP _IOWR(TE_IOCTL_MAGIC_NUMBER, 0x14, union te_cmd)
- #define TE_IOCTL_SS_CMD _IOR(TE_IOCTL_MAGIC_NUMBER, 0x30, int)
- #define TE_IOCTL_SS_CMD_GET_NEW_REQ 1
- #define TE_IOCTL_SS_CMD_REQ_COMPLETE 2
- #define OTE_MAX_DIR_NAME_LEN (64)
- #define OTE_MAX_FILE_NAME_LEN (128)
- #define OTE_MAX_DATA_SIZE (2048)
- #define OTE_RPMB_FRAME_SIZE 512
- #define SS_OP_MAX_DATA_SIZE 0x1000

Enumerations

- enum {
  TLK_SMC_REQUEST = 0xFFFF1000,
  TLK_SMC_GET_MORE = 0xFFFF1001,
  TLK_SMC_ANSWER = 0xFFFF1002,
  TLK_SMC_NO_ANSWER = 0xFFFF1003,
  TLK_SMC_OPEN_SESSION = 0xFFFF1004,
  TLK_SMC_CLOSE_SESSION = 0xFFFF1005 }

  Defines secure monitor calls (SMC) that clients use to communicate with trusted applications (TAs) in the secure world.
- enum {
  OTE_FILE_REQ_TYPE_CREATE = 0x1,
  OTE_FILE_REQ_TYPE_DELETE = 0x2,
  OTE_FILE_REQ_TYPE_OPEN = 0x3,
  OTE_FILE_REQ_TYPE_CLOSE = 0x4,
  OTE_FILE_REQ_TYPE_READ = 0x5,
  OTE_FILE_REQ_TYPE_WRITE = 0x6,
  OTE_FILE_REQ_TYPE_GET_SIZE = 0x7,
  OTE_FILE_REQ_TYPE_SEEK = 0x8,
  OTE_FILE_REQ_TYPE_TRUNC = 0x9,
  OTE_FILE_REQ_TYPE_RPMB_WRITE = 0x1001,
  OTE_FILE_REQ_TYPE_RPMB_READ = 0x1002 }

- enum {
  OTE_FILE_REQ_FLAGS_ACCESS_RO = 1,
  OTE_FILE_REQ_FLAGS_ACCESS_WO = 2,
  OTE_FILE_REQ_FLAGS_ACCESS_RW = 3 }
- enum {
  OTE_SEEK_WHENCE_SET = 1,
  OTE_SEEK_WHENCE_CUR = 2,
  OTE_SEEK_WHENCE_END = 3 }

7.4 ote_command.h File Reference

7.4.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Common Commands Description: Declares the common commands in the TLK interface.

Functions

- te_error_t te_open_session (te_session_t ∗session, te_service_id_t ∗service, te_operation_t ∗operation)
- void te_close_session (te_session_t ∗session)
- te_operation_t ∗ te_create_operation (void)
- void te_init_operation (te_operation_t ∗te_op)
- void te_deinit_operation (te_operation_t ∗teOp)
- te_error_t te_launch_operation (te_session_t ∗session, te_operation_t ∗te_op)
- void te_oper_set_command (te_operation_t ∗te_op, uint32_t command)
- void te_oper_set_param_int_ro (te_operation_t ∗te_op, uint32_t index, uint32_t Int)
- void te_oper_set_param_int_rw (te_operation_t ∗te_op, uint32_t index, uint32_t Int)
- void te_oper_set_param_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗base, uint32_t len)
- void te_oper_set_param_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗base, uint32_t len)
- void te_oper_set_param_persist_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗base, uint32_t len)
- void te_oper_set_param_persist_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗base, uint32_t len)
- uint32_t te_oper_get_command (te_operation_t ∗te_op)
- uint32_t te_oper_get_num_params (te_operation_t ∗te_op)
- te_error_t te_oper_get_param_type (te_operation_t ∗te_op, uint32_t index, te_oper_param_type_t ∗type)
- te_error_t te_oper_get_param_int (te_operation_t ∗te_op, uint32_t index, uint32_t ∗Int)
- te_error_t te_oper_get_param_mem (te_operation_t ∗te_op, uint32_t index, void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_persist_mem_ro (te_operation_t ∗te_op, uint32_t index, const void ∗∗base, uint32_t ∗len)
- te_error_t te_oper_get_param_persist_mem_rw (te_operation_t ∗te_op, uint32_t index, void ∗∗base, uint32-_t ∗len)
- void te_operation_reset (te_operation_t ∗te_op)

7.5 ote_common.h File Reference

7.5.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Common Declarations Description: Declares the common declarations in the TLK interface.

Data Structures

- struct te_service_id_t
- union te_session_t
- struct te_oper_param_t
- struct te_operation_t

Macros

- #define OTE_TASK_NAME_MAX_LENGTH 24
- #define OTE_TASK_PRIVATE_DATA_LENGTH 20

Typedefs

- typedef uint64_t cmnptr_t

Enumerations

- enum te_oper_param_type_t {
  TE_PARAM_TYPE_NONE = 0x0,
  TE_PARAM_TYPE_INT_RO = 0x1,
  TE_PARAM_TYPE_INT_RW = 0x2,
  TE_PARAM_TYPE_MEM_RO = 0x3,
  TE_PARAM_TYPE_MEM_RW = 0x4,
  TE_PARAM_TYPE_PERSIST_MEM_RO = 0x100,
  TE_PARAM_TYPE_PERSIST_MEM_RW = 0x101 }
- enum {
  TE_CRITICAL = 0,
  TE_ERR = 1,
  TE_INFO = 2,
  TE_SECURE = 3,
  TE_INTERFACE = 4,
  TE_RESULT = 5 }

Functions

- te_result_origin_t te_get_result_origin (te_session_t ∗session)
- int te_fprintf (int fd, char ∗fmt,...) __PRINTFLIKE(2

7.5.2 Enumeration Type Documentation

7.5.2.1 anonymous enum

Enumerator:

TE_CRITICAL

TE_ERR

TE_INFO

TE_SECURE

TE_INTERFACE

TE_RESULT

7.5.3 Function Documentation

7.5.3.1 int te_fprintf (int *fd,* char ∗ *fmt,* *...*)

For secure tasks: Redirects prints to Trusted Little Kernel (TLK) writes. This is a printf function for TLK services.

For clients: Prints to stdout.

7.6 ote_crypto.h File Reference

7.6.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Cryptography Description: Declares the cryptography APIs in the TLK.

Data Structures

- struct te_crypto_operation_info_t
- struct __te_crypto_operation_t
- struct te_crypto_rsa_key_t

Typedefs

- typedef struct __te_crypto_object ∗ te_crypto_object_t
- typedef struct
  __te_crypto_operation_t ∗ te_crypto_operation_t

Enumerations

- enum te_oper_crypto_algo_t {
  OTE_ALG_AES_ECB_NOPAD = 0x10000010,
  OTE_ALG_AES_CBC_NOPAD = 0x10000110,
  OTE_ALG_AES_CTR = 0x10000210,
  OTE_ALG_AES_CTS = 0x10000310,
  OTE_ALG_AES_ECB = 0x10000510,
  OTE_ALG_AES_CBC = 0x10000610,
  OTE_ALG_AES_CMAC_128 = 0x20000110,
  OTE_ALG_AES_CMAC_192 = 0x20000120,
  OTE_ALG_AES_CMAC_256 = 0x20000130,
  OTE_ALG_SHA_HMAC_224 = 0x20000210,
  OTE_ALG_SHA_HMAC_256 = 0x20000220,
  OTE_ALG_SHA_HMAC_384 = 0x20000230,
  OTE_ALG_SHA_HMAC_512 = 0x20000240,
  OTE_ALG_SHA_HMAC_1 = 0x20000250,
  OTE_ALG_RSA_PKCS_OAEP = 0x30000100,
  OTE_ALG_RSA_PSS = 0x30000200,
  OTE_ALG_PKCS1_Block1 = 0x30000300 }
- enum te_oper_crypto_algo_mode_t {
  OTE_ALG_MODE_ENCRYPT,
  OTE_ALG_MODE_DECRYPT,
  OTE_ALG_MODE_SIGN,
  OTE_ALG_MODE_VERIFY,
  OTE_ALG_MODE_DIGEST,
  OTE_ALG_MODE_DERIVE }

Functions

- te_error_t te_allocate_object (te_crypto_object_t ∗obj)
- te_error_t te_populate_object (te_crypto_object_t obj, te_attribute_t ∗attrs, uint32_t attr_count)
- void te_free_object (te_crypto_object_t obj)
- te_error_t te_allocate_operation (te_crypto_operation_t ∗oper, te_oper_crypto_algo_t algorithm, te_oper_-crypto_algo_mode_t mode)
- te_error_t te_set_operation_key (te_crypto_operation_t oper, te_crypto_object_t obj)
- te_error_t te_cipher_init (te_crypto_operation_t oper, void ∗iv, size_t iv_size)
- te_error_t te_cipher_update (te_crypto_operation_t oper, const void ∗src_data, size_t src_size, void ∗dst_-data, size_t ∗dst_size)
- te_error_t te_cipher_do_final (te_crypto_operation_t oper, const void ∗src_data, size_t src_len, void ∗dst_-data, size_t ∗dst_len)
- te_error_t te_rsa_init (te_crypto_operation_t oper)
- te_error_t te_rsa_handle_request (te_crypto_operation_t oper, const void ∗src_data, size_t src_size, void ∗dst_data, size_t ∗dst_size)
- void te_free_operation (te_crypto_operation_t oper)
- void te_generate_random (void ∗buffer, size_t size)
- te_error_t te_get_attribute_by_id (te_crypto_object_t object, te_attribute_id_t id, te_attribute_t ∗∗ret)

7.7 ote_error.h File Reference

7.7.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Error Handling Description: Declares the error handling codes for the TLK interface.

Enumerations

- enum te_error_t {
  OTE_SUCCESS = 0x00000000,
  OTE_ERROR_NO_ERROR = 0x00000000,
  OTE_ERROR_GENERIC = 0xFFFF0000,
  OTE_ERROR_ACCESS_DENIED = 0xFFFF0001,
  OTE_ERROR_CANCEL = 0xFFFF0002,
  OTE_ERROR_ACCESS_CONFLICT = 0xFFFF0003,
  OTE_ERROR_EXCESS_DATA = 0xFFFF0004,
  OTE_ERROR_BAD_FORMAT = 0xFFFF0005,
  OTE_ERROR_BAD_PARAMETERS = 0xFFFF0006,
  OTE_ERROR_BAD_STATE = 0xFFFF0007,
  OTE_ERROR_ITEM_NOT_FOUND = 0xFFFF0008,
  OTE_ERROR_NOT_IMPLEMENTED = 0xFFFF0009,
  OTE_ERROR_NOT_SUPPORTED = 0xFFFF000A,
  OTE_ERROR_NO_DATA = 0xFFFF000B,
  OTE_ERROR_OUT_OF_MEMORY = 0xFFFF000C,
  OTE_ERROR_BUSY = 0xFFFF000D,
  OTE_ERROR_COMMUNICATION = 0xFFFF000E,
  OTE_ERROR_SECURITY = 0xFFFF000F,
  OTE_ERROR_SHORT_BUFFER = 0xFFFF0010,
  OTE_ERROR_BLOCKED = 0xFFFF0011,
  OTE_ERROR_NO_ANSWER = 0xFFFF1003 }

  Defines Open Trusted Environment (OTE) error codes.

- enum te_result_origin_t {
  OTE_RESULT_ORIGIN_API = 1,
  OTE_RESULT_ORIGIN_COMMS = 2,
  OTE_RESULT_ORIGIN_KERNEL = 3,
  OTE_RESULT_ORIGIN_TRUSTED_APP = 4 }

  Defines the origin of an error.

7.8 ote_ext_nv.h File Reference

7.8.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Memory/Cache Management Description: Declares memory/cache management in the TLK.

Data Structures

- struct te_map_mem_addr_args_t

  Holds a pointer to the map memory for a specific OTE_CONFIG_MAP_MEM ID value.
- struct te_v_to_p_args_t

  Holds a pointer to the physical address for a specific virtual address.
- struct te_cache_maint_args_t

  Holds an op code and data used to for cache maintenance.

Enumerations

- enum te_ext_nv_cache_maint_op_t {
  OTE_EXT_NV_CM_OP_CLEAN = 1,
  OTE_EXT_NV_CM_OP_INVALIDATE = 2,
  OTE_EXT_NV_CM_OP_FLUSH = 3 }
- enum te_ss_config_option_t {
  OTE_SS_CONFIG_RPMB_ENABLE = 0x0000001,
  OTE_SS_CONFIG_CPC_ENABLE = 0x0000002 }

Functions

- te_error_t te_ext_nv_cache_maint (te_ext_nv_cache_maint_op_t op, void ∗addr, uint32_t length)
- te_error_t te_ext_nv_virt_to_phys (void ∗addr, uint64_t ∗paddr)
- te_error_t te_ext_nv_get_map_addr (uint32_t id, void ∗∗addr)

7.9 ote_ioctl.h File Reference

7.9.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: IOCTL Values Description: Declares TLK IOCTL values.

Enumerations

- enum {
  OTE_IOCTL_GET_MAP_MEM_ADDR = 1UL,
  OTE_IOCTL_V_TO_P = 2UL,
  OTE_IOCTL_CACHE_MAINT = 3UL,
  OTE_IOCTL_TA_TO_TA_REQUEST = 4UL,
  OTE_IOCTL_GET_PROPERTY = 5UL,
  OTE_IOCTL_GET_DEVICE_ID = 6UL,
  OTE_IOCTL_GET_TIME_US = 7UL,
  OTE_IOCTL_GET_RAND32 = 8UL,
  OTE_IOCTL_SS_REQUEST = 9UL,
  OTE_IOCTL_TASK_REQUEST = 10UL,
  OTE_IOCTL_SS_GET_CONFIG = 11UL,
  OTE_IOCTL_REGISTER_TA_EVENT = 12UL,
  OTE_IOCTL_TASK_PANIC = 13UL }

  Defines IOCTL syscall interface parameters.

7.10 ote_manifest.h File Reference

7.10.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Manifest Layout Description: Declares the layout of the manifest in the TLK.

Data Structures

- struct OTE_MANIFEST

Macros

- #define OTE_CONFIG_MIN_STACK_SIZE(sz) OTE_CONFIG_KEY_MIN_STACK_SIZE, sz
- #define OTE_CONFIG_MIN_HEAP_SIZE(sz) OTE_CONFIG_KEY_MIN_HEAP_SIZE, sz
- #define OTE_CONFIG_MAP_MEM(id, off, sz) OTE_CONFIG_KEY_MAP_MEM, id, off, sz
- #define OTE_CONFIG_RESTRICT_ACCESS(clients) OTE_CONFIG_KEY_RESTRICT_ACCESS, clients
- #define OTE_CONFIG_AUTHORIZE(perm) OTE_CONFIG_KEY_AUTHORIZE, perm
- #define OTE_CONFIG_TASK_INITIAL_STATE(state) OTE_CONFIG_KEY_TASK_ISTATE, state
- #define OTE_MANIFEST_ATTRS __attribute((aligned(4))) __attribute((section(".ote.manifest")))

Enumerations

- enum ote_config_key_t {
  OTE_CONFIG_KEY_MIN_STACK_SIZE = 1,
  OTE_CONFIG_KEY_MIN_HEAP_SIZE = 2,
  OTE_CONFIG_KEY_MAP_MEM = 3,
  OTE_CONFIG_KEY_RESTRICT_ACCESS = 4,
  OTE_CONFIG_KEY_AUTHORIZE = 5,
  OTE_CONFIG_KEY_TASK_ISTATE = 6 }
- enum {
  OTE_RESTRICT_SECURE_TASKS = 1 << 0,
  OTE_RESTRICT_NON_SECURE_APPS = 1 << 1 }
- enum { OTE_AUTHORIZE_INSTALL = 1 << 10 }

- enum {
  OTE_MANIFEST_TASK_ISTATE_IMMUTABLE = 1 << 0,
  OTE_MANIFEST_TASK_ISTATE_STICKY = 1 << 1,
  OTE_MANIFEST_TASK_ISTATE_BLOCKED = 1 << 2 }

7.11 ote_memory.h File Reference

7.11.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Memory Functions Description: Declares the memory functions in the TLK.

Functions

- void ∗ te_mem_alloc (uint32_t size)
- void ∗ te_mem_calloc (uint32_t size)
- void ∗ te_mem_realloc (void ∗buffer, uint32_t size)
- void te_mem_free (void ∗buffer)
- void te_mem_fill (void ∗buffer, uint32_t value, uint32_t size)
- void te_mem_move (void ∗dest, const void ∗src, uint32_t size)
- int te_mem_compare (const void ∗buffer1, const void ∗buffer2, uint32_t size)

7.12 ote_nvcrypto.h File Reference

7.12.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: NVIDIA Cryptography Description: Declares the cryptography APIs in the TLK.

Functions

- te_error_t ote_nvcrypto_init (void)
- te_error_t ote_nvcrypto_deinit (void)
- te_error_t ote_nvcrypto_get_keybox (uint32_t keybox_index, void ∗buf, uint32_t ∗len)
- te_error_t ote_nvcrypto_get_storage_key (uint8_t ∗key, uint32_t key_size)

  Gets the storage key.
- te_error_t ote_nvcrypto_get_rollback_key (uint8_t ∗key, uint32_t key_size)

  Gets the rollback key.

7.13 ote_otf.h File Reference

7.13.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: On-the-Fly (OTF) Decoder Service Description: Declares functions for the TLK OTF decoder service.

Functions

- te_error_t ote_otf_init (te_session_t ∗∗otfSession)

  Initializes the on-the-fly (OTF) hardware.
- te_error_t ote_otf_deinit (te_session_t ∗∗otfSession)

  Resets the OTF hardware and erases any previous keys.
- te_error_t ote_otf_setkey (void ∗buffer, uint32_t len, uint32_t ∗keySlot, te_session_t ∗otfSession)

  Sets the key to be used by the OTF hardware.
- te_error_t ote_otf_set_key_at (void ∗buffer, uint32_t len, uint32_t keySlot, te_session_t ∗otfSession)

  Sets the key to be used by the OTF hardware at the specified slot.
- te_error_t ote_otf_erasekey (te_session_t ∗otfSession)

  Erases keys from the OTF hardware.

7.14 ote_rtc.h File Reference

7.14.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Real-Time Clock (RTC) Services Description: Declares common declarations and functions for the TLK RTC service.

Functions

- te_error_t ote_rtc_init (void)

  Initializes the RTC hardware.
- te_error_t ote_rtc_deinit (void)

  Resets the RTC hardware and erases any previous keys.
- te_error_t ote_rtc_get_time (uint32_t ∗rtc)

  Gets the RTC from the RTC hardware.

7.15 ote_service.h File Reference

7.15.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Service Interface Description: Declares data types and functions for the TLK services.

Data Structures

- struct te_request_t

  Holds the layout of the `te_oper_param_t` structures which must match the layout sent in by the non-secure (NS) world via the TrustZone Secure Monitor Call (TZ SMC) path.
- struct te_ta_to_ta_request_args_t
- struct te_entry_point_message_t
- struct te_identity_t
- struct te_get_property_args_t
- struct te_device_unique_id
- struct te_panic_args_t
- struct ta_event_args_t

Macros

- #define DEVICE_UID_SIZE_BYTES 16
- #define OTE_PANIC_MSG_MAX_SIZE 128
- #define OTE_TE_FPRINTF_PREFIX_MAX_LENGTH (OTE_TASK_NAME_MAX_LENGTH + 4)

Typedefs

- typedef te_error_t(∗ ta_event_handler_t)(ta_event_args_t ∗args)

Enumerations

- enum {
  CREATE_INSTANCE = 1UL,
  DESTROY_INSTANCE = 2UL,
  OPEN_SESSION = 3UL,
  CLOSE_SESSION = 4UL,
  LAUNCH_OPERATION = 5UL,
  TA_EVENT = 6UL }
- enum {
  TE_LOGIN_PUBLIC = 0,
  TE_LOGIN_TA = 7 }
- enum {
  TE_PROP_DATA_TYPE_UUID = 1,
  TE_PROP_DATA_TYPE_IDENTITY = 2 }
- enum te_property_type_t {
  TE_PROPERTY_CURRENT_TA = 0xFFFFFFFF,
  TE_PROPERTY_CURRENT_CLIENT = 0xFFFFFFFE,
  TE_PROPERTY_TE_IMPLEMENTATION = 0xFFFFFFFD }
- enum ta_event_id {
  TA_EVENT_RESTORE_KEYS = 0,
  TA_EVENT_MASK = (1 << TA_EVENT_RESTORE_KEYS) }

Functions

- void te_exit_service (void)
- te_error_t te_init (int argc, char ∗∗argv)
- void te_destroy (void)
- te_error_t te_create_instance_iface (void)
- void te_destroy_instance_iface (void)
- te_error_t te_open_session_iface (void ∗∗sctx, te_operation_t ∗oper)
- void te_close_session_iface (void ∗sctx)
- te_error_t te_receive_operation_iface (void ∗sctx, te_operation_t ∗oper)
- void ∗ ote_get_instance_data (void)
- void ote_set_instance_data (void ∗sessionContext)
- te_error_t te_get_current_ta_uuid (te_service_id_t ∗value)
- te_error_t te_get_client_ta_identity (te_identity_t ∗value)
- te_error_t te_get_client_identity (te_identity_t ∗value)
- te_error_t te_get_device_unique_id (te_device_unique_id ∗uid)
- void te_panic (char ∗msg) __attribute__((noreturn))
- void te_fprintf_set_prefix (const char ∗prefix)
- void te_oper_dump_param (te_oper_param_t ∗param)
- void te_oper_dump_param_list (te_operation_t ∗te_op)
- te_error_t te_register_ta_event_handler (ta_event_handler_t handler, uint32_t events_mask)

7.16 ote_storage.h File Reference

7.16.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Storage Services Description: Declares data types and functions for TLK storage services.

Macros

- #define TE_STORAGE_OBJID_MAX_LEN 64

Typedefs

- typedef struct
  __te_storage_object ∗ te_storage_object_t

Enumerations

- enum te_storage_flags_t {
  OTE_STORAGE_FLAG_ACCESS_READ = 0x1,
  OTE_STORAGE_FLAG_ACCESS_WRITE = 0x2,
  OTE_STORAGE_FLAG_ACCESS_WRITE_META = 0x4 }
- enum te_storage_whence_t {
  OTE_STORAGE_SEEK_WHENCE_SET = 0x1,
  OTE_STORAGE_SEEK_WHENCE_CUR = 0x2,
  OTE_STORAGE_SEEK_WHENCE_END = 0x3 }

Functions

- te_error_t te_create_storage_object (char ∗name, te_storage_flags_t flags, te_storage_object_t ∗obj)
- te_error_t te_open_storage_object (char ∗name, te_storage_flags_t flags, te_storage_object_t ∗obj)
- te_error_t te_read_storage_object (te_storage_object_t obj, void ∗buffer, uint32_t size, uint32_t ∗count)
- te_error_t te_write_storage_object (te_storage_object_t obj, const void ∗buffer, uint32_t size)
- te_error_t te_get_storage_object_size (te_storage_object_t obj, uint32_t ∗size)
- te_error_t te_seek_storage_object (te_storage_object_t obj, int32_t offset, te_storage_whence_t whence)
- te_error_t te_trunc_storage_object (te_storage_object_t obj, uint32_t size)
- te_error_t te_delete_storage_object (te_storage_object_t obj)
- te_error_t te_delete_named_storage_object (const char ∗name)
- te_error_t te_close_storage_object (te_storage_object_t obj)

7.17 ote_task_load.h File Reference

7.17.1 Detailed Description

NVIDIA Trusted Little Kernel Interface: Task-Loading Interface Description: Declares data types and functions for the TLK task-loading interface.

Data Structures

- struct te_app_load_memory_request_args_t
- struct te_task_info_t
- struct te_app_prepare_args_t
- struct te_task_restrictions_t
- struct te_app_start_args_t
- struct te_app_list_args_t
- struct te_list_apps
- struct te_get_task_info_t
- struct te_memory_mapping_t
- struct te_get_task_mapping_t
- struct te_get_pending_map_args_t
- struct te_system_info_args_t
- struct te_app_unload_args_t
- struct te_app_unload_t
- struct te_app_block_args_t
- struct te_app_block_t
- struct te_task_request_args_s

Typedefs

- typedef struct te_list_apps te_list_apps_t
- typedef struct
  te_task_request_args_s te_task_request_args_t

Enumerations

- enum install_type_t {
  INSTALL_TYPE_NORMAL = 0,
  INSTALL_TYPE_REJECT,
  INSTALL_TYPE_UPDATE }

  Defines install types for use with te_app_start() and te_app_start_args_t.
- enum te_get_info_type_t {
  OTE_GET_TASK_INFO_REQUEST_INDEX,
  OTE_GET_TASK_INFO_REQUEST_UUID,
  OTE_GET_TASK_INFO_REQUEST_SELF }
- enum te_app_id_t {
  OTE_APP_ID_INDEX,
  OTE_APP_ID_UUID }
- enum te_task_opcode_t {
  OTE_TASK_OP_UNKNOWN,
  OTE_TASK_OP_MEMORY_REQUEST,
  OTE_TASK_OP_PREPARE,
  OTE_TASK_OP_START,
  OTE_TASK_OP_LIST,
  OTE_TASK_OP_GET_TASK_INFO,
  OTE_TASK_OP_SYSTEM_INFO,
  OTE_TASK_OP_PENDING_MAPPING,
  OTE_TASK_OP_UNLOAD,
  OTE_TASK_OP_BLOCK,
  OTE_TASK_OP_UNBLOCK,
  OTE_TASK_OP_GET_MAPPING }

Functions

- te_error_t te_app_request_memory (u_int app_size, uintptr_t ∗app_addr, uint32_t ∗app_handle)
- te_error_t te_app_prepare (uint32_t app_handle, te_task_info_t ∗app_task_info)
- te_error_t te_app_start (uint32_t app_handle, install_type_t install_type, te_task_restrictions_t ∗app_-restrictions)
- te_error_t te_list_apps (te_list_apps_t ∗app_list)
- te_error_t te_task_get_info (te_get_task_info_t ∗task_info)
- te_error_t te_task_get_mapping (te_get_task_mapping_t ∗task_mapping)
- te_error_t te_get_pending_task_mapping (uint32_t app_handle, te_memory_mapping_t ∗app_map)
- te_error_t te_task_get_name_self (char ∗name, uint32_t ∗len_p)
- te_error_t te_task_system_info (uint32_t type)
- te_error_t te_app_unload (te_app_unload_t ∗arg_unload)
- te_error_t te_app_block (te_app_block_t ∗app)
- te_error_t te_app_unblock (te_app_block_t ∗app)