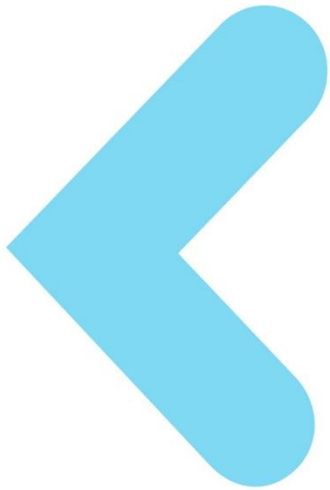
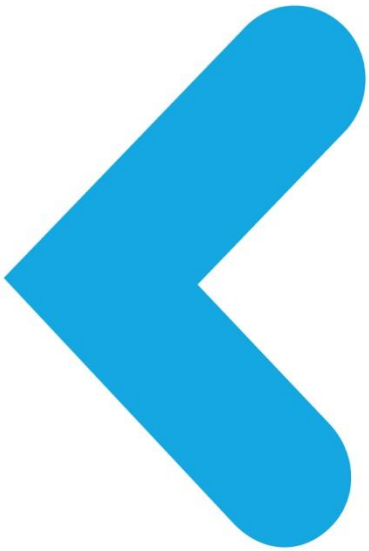
TRUSTONIC



Kinibi Integration Guide



PREFACE

DOCUMENT HISTORY

Date	Modification
January 5th, 2013	First version
May 22nd, 2013	Update for Kinibi-202
November 21st, 2013	Updated for Kinibi-300
June 9th, 2014	Updated for Kinibi-301
December 1st, 2014	Updated for Kinibi-302A: Integration information for Android 64 bit OS added Android CTS waiver information and SEAndroid information added. General improvements. Fix the paths of the Tui components. Remove sections about DrSecureStorage Detail on Registry and Factory Reset. Added missing information for the location of the keymaster in the Android File System. Fixed the error codes returned by the DRM driver. Security warnings added in TUI integration sections
February 19th, 2015	Updated for Kinibi-303A Updated permissions for the mcRegistry directories. TUI normal world architecture change to optimize memory consumption.
August 12th, 2015	Updated for Kinibi-310A: - Remove kernel module `MobicoreKernelApi` - Update kernel Makefile and Kconfig options - Add ueventd information for TUI driver - Add a new debugfs directory `trustonic_tee` for system debugging - Document FIQ forward
October 27th, 2015	Updated for Kinibi-310B: - Add Android SE Proxy - Add Android Keymaster M and Gatekeeper support - New entries in debugFS - IRAM support

February 9th, 2016	Updated for Kinibi-310C: - Add standalone Android SE Proxy
March 30th, 2016	Updated for Kinibi-311A: - Add Android M Verified Boot - Add Rollback Protected Storage - Add tee-ps tool for performance analysis and post-mortem debug -Add missing TUI HAL functions
May 19th, 2016	Fixed the number of instances of the Keymaster TA.
May 24th, 2016	Remove all mentions of TeeAuthService
May 30th, 2016	Add information about the Normal World daemon's light mode for recovery
June 6th, 2016	The number of threads of the TUI driver can be 5 or 6.
July 27th, 2016	Update for Android Nougat
September 9th, 2016	Added guidance on checking the Kinibi version Updated the Chapter on Validating the product to include Kinibi version verification check.
September 28th, 2016	Updated for Kinibi-400A: - Add System TA Downgrade Protection to Rollback Protection - Changes in NWd for access to /efs and multi-partition support - Add TEE Image Builder - Add ATF fastcalls for GlobalPlatform compliance

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This document explains how to integrate Trustonic's Kinibi Trusted Operating System in to a hardware platform. Kinibi integration consists of:

- Integrating the Kinibi Normal World components in Google's Android OS.
- Integrating the Kinibi Secure World components.
- Integrating Trustonic's Android Keymaster modules for enhanced security.
- Implementing support of pluggable DRM framework.
- Implementing Kinibi Trusted User Interface (TUI).
- Implementing RPMB Driver or Monotonic Counter TA.
- Running and passing the Trustonic Test Suite to verify the integration.

Note that in order to enable Kinibi-4xx and the OTA-Operated Containers on a Device, the device manufacturer must install and integrate Trustonic's Key Provisioning Host (KPH) into the manufacturing line. The KPH is a tool that injects the TEE Authentication Token into the device, storing a copy in the Trustonic backend system, Trustonic Directory. For details of this process and how to integrate the KPH please consult the document entitled *Kinibi_Provisioning_Manual.pdf*.

The Kinibi product package is structured as follows:

- `/Documentation`, this directory contains documentation detailing how to integrate Kinibi into a device and how to use Kinibi provisioning tools.
- `/AndroidIntegration`, this directory contains all the Normal World components to be integrated into Android.
- `/LinuxIntegration`, this directory contains all the Normal World components to be integrated into Linux.
- `/SecureIntegration`, this directory contains all the Secure World components including the Kinibi core binary and Trustonic System Trusted Applications.
- `/t-base-dev-kit`, the Kinibi software development kit (SDK) with documentation and Samples for Trusted Applications developers.
- */TTS*, a comprehensive test suite allowing integrators to quickly validate that all the critical features of Kinibi have been successfully integrated.

2 KINIBI NORMAL WORLD COMPONENTS

Trustonic's Kinibi OS is supplied with a number of components that need integrating into the Android image. These Normal World (NWd) components are:

In Android User space:

- The Kinibi Daemon: in charge of filesystem access for the Secure World Around two axes: secure filesystem and Trustonic registry.
- The Kinibi Proxy: provides local socket access to the Secure World when the driver access is denied.
- The Root Provisioning Agent (RootPA): used to install Service Provider Trusted Applications. The RootPA includes the Curl library, for XML manipulations and a log library.
- The Kinibi Client library: A selection of user space APIs available for Normal World application developers. Tries to use the driver for Secure World access and falls back to using the proxy on access denied.
- The Kinibi registry interface: API's for the management of the Kinibi registry.
- The Kinibi Keymaster library: An API to interface with Android's Keymaster and Gatekeeper feature.

In Linux Kernel space:

- The Kinibi Linux driver: A component in charge of communication between the Normal World user space client and the Secure World. There are two interfaces available: one reserved for the Kinibi Daemon and one allocated to direct communication with all the user space clients.
- The Kinibi Linux driver kernel API: kernel level APIs available to Normal World driver developers.

All Normal World components for the Kinibi product are provided in source code and binary forms, with support for 32-bit and 64-bit Android versions and 32-bit Linux, in the `/AndroidIntegration` directory for Android, in the `/LinuxIntegration` directory for Linux . The purpose of this folder is to provide a simple way of integrating the Kinibi Normal World components into an existing Android source tree.

2.1 BUILDING NORMAL WORLD COMPONENTS

For Android, Source Code the source code can be found in */AndroidIntegration/Src*.

For generic Linux, the Source Code can be found in */LinuxIntegration/Src*.

This Source Code folder contains two sub-folders:

- A `mobicore` sub-folder which contains the Android user space components (executables and libraries).
- A `gud` sub-folder which contains the Linux kernel components.

2.1.1 Kinibi User-space Components

2.1.1.1 Android Compilation

The standard way of building the Kinibi user space components is to use a complete Android source tree. Simply copy the `mobicore` folder into the Android source tree `external` folder.

You should end up with the following files and folders:

```
Android root/
    external/
      mobicore/
        ClientLib/
        curl/
        Daemon/
        rootpa/
        tlcm/
        TlcTeeGatekeeper/
        TlcTeeKeymaster/
        TlcTeeKeymasterM/
        TuiService/
        Android.mk
```

Afterwards you have to delete the components that are not adequate for your Android version:

- TlcTeeGatekeeper and TlcTeeKeymasterM are only required for Android 6.0 Marshmallow
- TlcTeeKeymaster is only for Android versions up to and including 5.x Lollipop

To build the Kinibi components, simply launch an Android tree full build.

2.1.1.1.1 Keymaster1.0 and Gatekeeper

To build the Keymaster1.0 and Gatekeeper components, you have to modify your device.mk, e.g.

device/goldfish/device.mk

```
PRODUCT_PACKAGES += \
     gatekeeper.goldfish \
     keystore.goldfish
```

This works because in our makefiles we set the following.

external/mobicore/TlcTeeKeymasterM/Android.mk

```
LOCAL_MODULE := keystore.$(TARGET_BOARD_PLATFORM)
```

external/mobicore/TlcTeeGatekeeper/Android.mk

```
LOCAL_MODULE := gatekeeper.$(TARGET_BOARD_PLATFORM)
```

2.1.1.2 Linux Compilation

Linux compilation is based on CMake. In each component sub-folder, you can find a CMakeLists.txt file that contains the rules to build the Kinibi components, as shown below:

```
mobicore/
├──── ClientLib
│     ├──── CMakeLists.txt
│     ├──── include
│     └──── src
└──── Daemon
      ├──── CMakeLists.txt
      ├──── include
      ├──── src
      └──── systemd
```

For the Daemon we also provide a system (the standard Linux init program) script to launch the daemon service on boot.

For the ClientLib, CMake generates a pkgconfig to be used by packages that depend on it (pkgconfig files contain the paths to the include and lib directory).

On your PC, you want to install CMake, pkgconfig and your arm toolchain used to cross-compile the component.

Information about CMake variables:

- CMAKE_PROJECT_NAME : defines the name used in the pkgconfig files
- CMAKE_C_COMPILER: path to your cross-compiler.
- CMAKE_CXX_COMPILER: path to your cross-compiler.
- CMAKE_INSTALL_PREFIX: path where cmake installs the compilation result (headers, libs and pkgconfig files). You should use the same folder for ClientLib and Daemon compilation.
- TEE_VERSION: defines the component version in pkgconfig files, should be aligned on Kinibi version (311A, 311B, …).
- PKG_CONFIG_PATH: path where libs and bins are installed, same folder as CMAKE_INSTALL_PREFIX with pkgconfig folder.

Configure pkgconfig for finding dependencies :

```
export PKG_CONFIG_PATH=~/install/lib/pkgconfig/
```

2.1.1.2.1 ClientLib

For example, go into `mobicore/ClientLib/` folder then:

```
cmake . -DCMAKE_PROJECT_NAME=ClientLib -DCMAKE_C_COMPILER=arm-linux-
gnueabihf-gcc -DCMAKE_CXX_COMPILER=arm-linux-gnueabihf-g++
-DCMAKE_INSTALL_PREFIX=~/install/ -DTEE_VERSION=XXX -Wno-dev
make && make install
```

2.1.1.2.2 Daemon

For example, go into `mobicore/Daemon/` folder then:

```
cmake . -DCMAKE_PROJECT_NAME=McRegistry -DCMAKE_C_COMPILER=arm-linux-
gnueabihf-gcc -DCMAKE_CXX_COMPILER=arm-linux-gnueabihf-g++
-DCMAKE_INSTALL_PREFIX=~/install/ -DTEE_VERSION=XXX -Wno-dev
make && make install
```

2.1.2 Kinibi Kernel-space Components

The Linux driver directory `gud` within the Kinibi package is designed to be directly dropped in the Linux kernel source tree, specifically the `drivers` folder.

It contains one modules:

- The Kinibi Linux driver.

Once the `gud` directory has been copied, you should have the following organization:

```
Linux kernel/
    drivers/
        gud/
            Kconfig/
            Makefile/
            MobicoreDriver/
            TlcTui/
```

Please note that copying the `gud` directory to a different folder within the Linux source tree will not work.

To enable automatic building inside the kernel follow these steps:

- Update the Linux kernel configuration file `linux/drivers/Kconfig` with the following line which must be inserted before the last `endmenu` entry:

  ```
  source "drivers/gud/Kconfig"
  ```

- Add the Kinibi Linux driver to the Linux kernel build file `linux/drivers/Makefile` :

  ```
  obj-y += gud/
  ```

- Run `make menuconfig` and in the `Device Drivers` page of the configuration menu, please select:
  - `Trustonic TEE Driver`

  ```
  <*> Trustonic TEE Driver
  [ ]   Trustonic TEE uses LPAE
  [ ]   Trustonic TEE driver debug mode
  [ ]   Trustonic Trusted UI
  [ ]  Trustonic Trusted UI with fb_blank
  ```

- Finally, run `make` again and the Kinibi kernel components will be included in the kernel image.

2.2 INTEGRATION IN ANDROID (32 AND 64 BIT)

2.2.1 Integration in Device File System

The tables below give the directories in the Android file system where Kinibi components must be placed.

2.2.1.1 32 bit Android OS file system:

Component	Name of the binary	Containing folder
Kinibi Daemon	`mcDriverDaemon`	`/vendor/bin/`
Kinibi Proxy	`trustonic_tee_proxy`	`/vendor/bin/`
Root Provisioning Agent	`RootPA.apk`	`/system/app/`
Kinibi Client library	`libMcClient.so`	`/vendor/lib/`
Kinibi registry interface	`libMcRegistry.so`	`/vendor/lib/`
KeyMaster library	`libMcTeeKeymaster.so`	`/system/lib/`
Keymaster 1.0 library	`keystore.$DEVICE.so`	`/vendor/lib/hw/`
Gatekeeper library	`gatekeeper.$DEVICE.so`	`/vendor/lib/hw/`
Curl library	`libcurl.so`	`/system/lib/`
RootPA Native library	`libcommonpawrapper.so`	`/system/lib/`

2.2.1.2 64 bit Android OS file system:

Component	Name of the binary	Containing folder
Kinibi Daemon *64 bit*	`mcDriverDaemon`	`/vendor/bin/`
Kinibi Prox *64 bit*	`trustonic_tee_proxy`	`/vendor/bin/`
Root Provisioning Agent *32 bit*	`RootPA.apk`	`/system/app/`
Kinibi Client library *32 and 64 bit*	`libMcClient.so`	`/vendor/lib/`, `/vendor/lib64/`
Kinibi registry interface *32 and 64 bit*	`libMcRegistry.so`	`/vendor/lib/`, `/vendor/lib64/`
Keymaster library *32 and 64 bit*	`libMcTeeKeymaster.so`	`/system/lib/`, `/system/lib64/`

Keymaster 1.0 library *32 and 64 bit*	`keystore.$DEVICE.so`	`/vendor/lib/hw/`, `/vendor/lib64/hw/`
Gatekeeper library *32 and 64 bit*	`gatekeeper.$DEVICE.so`	`/vendor/lib/hw/`, `/vendor/lib64/hw/`
RootPA Native library *32 bit*	`libcommonpawrapper.so`	`/system/lib/`

The RootPA is a standard Android app and needs to be signed like any non-system apps (the version provided by Trustonic is not signed).

2.2.1.3 32 bit and 64 bit Android OS file system

If the Kinibi Linux driver modules are not built within the Linux kernel tree you must place them in the following locations:

Component	Name of the binary	Containing folder
Kinibi user device	`mcDrvModule.ko`	`/system/lib/modules/`

A couple of Trusted Applications have to be placed into the registry folder:

Component	Name of the binary	Containing folder
Content Management Trusted Application	`07010000…0000.tlbin` *`(tlcm.axf)`*	`/vendor/app/mcRegistry`
Keymaster TA	`07060000…0000.tlbin` *`(tlTeeKeymaster.axf)`*	`/vendor/app/mcRegistry`
Keymaster 1.0 TA	`07060000…004D.tlbin` *`(tlTeeKeymasterM.axf)`*	`/vendor/app/mcRegistry`
Gatekeeper TA	`07061000…0000.tlbin` *`(tlTeeGatekeeper.axf)`*	`/vendor/app/mcRegistry`

In addition, since Android N (7.0), in order to support 3rd party Trusted Application (Service provider TA) accessing the TEE interfaces, it is mandatory for the integrator to include the `libMcClient.so` and `libMcRegistry.so` in the list of Android vendor specific public libraries:

```
/vendor/etc/public.libraries.txt
```

2.2.2 Permissions and Access Rights

To enable the correct operation of Kinibi and any Trusted Applications it is mandatory to ensure all Kinibi components have the correct permissions and access rights. The following section highlights the required settings.

- Kinibi Daemon (`mcDriverDaemon`):
  The Kinibi Daemon should have permissions `0755`.

- Kinibi libraries (`libMc*, libcurl, libcommonpawrapper`):
  No specific permissions are needed for libraries.
- Kinibi Linux driver:
  The Linux driver needs two device nodes in `/dev` for the two different access mechanisms: `/dev/mobicore` for administration purpose, and `/dev/mobicore-user` for client access. Permissions for these two nodes must be defined in the following way:

```
/dev/mobicore-user 0666 system:system
/dev/mobicore      0600 system:system
```

  This should be done by adding the following lines to your ueventd.<hardware>.rc file:

```
# Trustonic TEE
/dev/mobicore      0600 system system
/dev/mobicore-user 0666 system system
```

  Note: `/dev/mobicore-user` is only created when the TEE has started successfully, which is triggered by the first daemon's connection to `/dev/mobicore`.

Trustonic have been granted a waiver by Google for the permission settings of /dev/mobicore-user in order to pass their Compliance Test Suite (CTS).

Details of the review and acceptance by Google can be found at
https://android-review.googlesource.com/#/c/63940/

For more information on the reasons behind the permission requirements of `/dev/mobicore-user` please consult Appendix 1.

2.2.3 Directories Requirements

There are two directories that must be created in the device file-system for Kinibi. These paths are used by Kinibi components at runtime.

2.2.3.1 mcRegistry directory

This directory is used by Kinibi to store non-permanent data that can be deleted with a factory reset or device wipe.

The mcRegistry directory stores content relating to Device Binding, including:

- TEE Authentication Token (also known as the AuthToken)
- TEE Authentication Token backup (also known as the AuthToken backup)

Note that the TEE Authentication Token backup is an exception, and it must be preserved across factory reset or device wipe (see section **Error! Reference source not found.**)

The mcRegistry directory also stores content relating to the OTA-management of third-party content, including:

- Kinibi Root Containers
- Service Provider Containers
- Service Provider TA Containers
- Service Provider TA binaries (used only for TAs using GP APIs)

Service Provider and System TAs making use of GP Trusted Storage APIs automatically store Secure Objects in sub-folders of the mcRegistry directory.

The mandatory path is:

```
/data/misc/mcRegistry
```

Permissions should be as follows:

- User: `system`, can read/write/execute
- Group: `system`, can read/write/execute
- Others can read/execute

In UNIX speak, permissions `0775`, user/group `system:system`.

2.2.3.2 Persistent Trusted Applications and Secure Drivers

Trusted Applications and Secure Drivers can be stored anywhere in the device file system. However, Persistent Trusted Applications or Secure Driver binaries must be placed in a non-volatile partition of the file system in order to survive a factory reset or device wipe.

Trustonic System Trusted Applications like the Content Management Trusted Application (CMTL) must be found in:

```
/vendor/app/mcRegistry
```

Permissions should be as follows:

- User: `system`, can read/write/execute
- Group: `system`, can read/execute
- Others can read/execute

In UNIX speak, permissions `0755`, user/group `system:system`.

Please note that there is no content generated here at runtime by Kinibi, the directory is only used to store persistent System Trusted Applications.

2.2.3.3 Persistent read-write directory (/efs)

Kinibi needs to store system data in a location that is persistent across factory reset or device wipe.

This requirement presently applies only to the following two files:

- TEE Authentication Token backup (also known as the AuthToken backup)
- Secure Storage Partition 1 used for System TA downgrade protection (see 9.4)

Since locations that are retained permanently are often highly OEM-specific, the OEM has to customize this location. A common location for this directory is:

```
/efs
```

2.2.3.3.1 AuthToken backup

The OEM has to customize the behavior. There is presently no environment variable governing the location of this container, and persistence is often achieved by restoring the container at each device boot, or as part of the factory reset procedure.

Note that, for implementation reasons, while the Containers held in this location must be persistent across factory reset and device wipe, there is a requirement for Kinibi to occasionally write these Containers at runtime.

2.2.3.3.2 Persistent mcRegistry Directory

The Kinibi daemon needs access to a persistent mcRegistry directory for the System TA downgrade protection to work.

A possible path is:

```
/efs/misc/mcRegistry
```

Permissions should be as follows:

- User: `system`, can read/write/execute
- Group: `system`, can read/write/execute
- Others can read/execute

In UNIX speak, permissions `0775`, user/group `system:system`.

The OEM must provide this path as argument -P1 <path> on the command line when starting the Kinibi daemon.

2.2.4 Proxy

For systems where downloaded applications do not have permission to open the Linux driver device, there is a separate proxy service that it accessible via a socket. The selection is done automatically in the TEE system client library (ClientLib), so no application change is required.

2.2.5 Starting up Kinibi

2.2.5.1 Standard mode

To start the Kinibi daemon and proxy services automatically at start-up, the following lines must be added to one of the `/init.*.rc` files on the Android device:

```
service trustonic-daemon /vendor/bin/mcDriverDaemon -P1 /efs/misc/mcRegistry
    class core
    user system
    group system

service trustonic-proxy /vendor/bin/trustonic_tee_proxy
    class core
    user system
    group system
```

The daemon must have system permissions for `user` and `group` otherwise the OEM is responsible to set the correct parameters for the service. The proxy must be user system to be accepted by the driver.

The OEM is also responsible for deciding how the system should behave in the event of an error, for example, if the device fails to correctly initialize Kinibi or if the Kinibi daemon crashes.

2.2.5.2 Recovery mode

In recovery mode, the proxy is not expected to be needed. Also, some services such as Secure File System and Trusted UI are not necessary. The daemon can thus be started without those two services, in “light” mode:

```
service trustonic-daemon /vendor/bin/mcDriverDaemon -l
    class core
    user system
    group system
```

2.2.6 SEAndroid Configuration

2.2.6.1 SEAndroid interaction with Kinibi

As the Normal-World components for Kinibi can be configured by the OEM, the directories and naming of binaries in the figure below are mostly instructive, and can vary between installations. The exact configuration is left to the OEM although the permissions must be implemented correctly for normal OTA operation. It is essential that after any policy change the Kinibi Test Suite is executed to ensure there are no adverse effects on application provisioning via Trustonic's Directory servers.

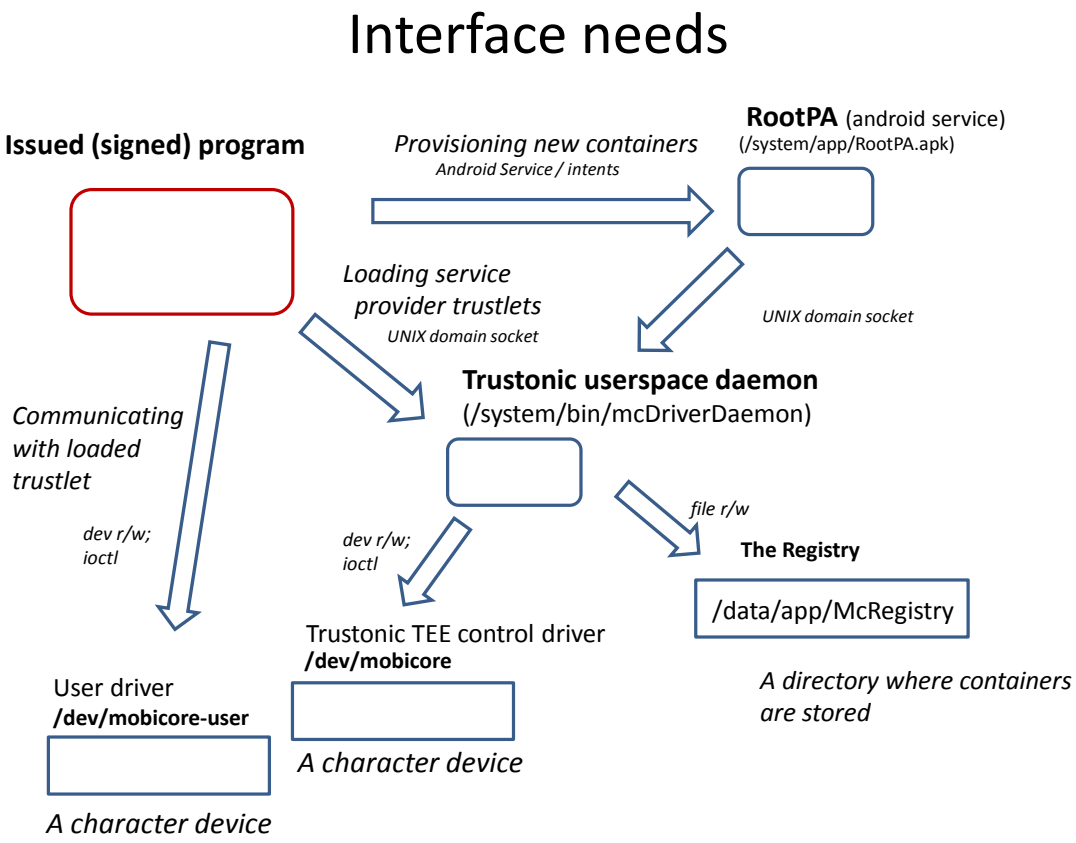


Figure 1: SEAndroid interface requirements

The main flow of program installation (from a permission perspective) is as follows and depicted in the figure above. The Android program X (downloaded from a store) leveraging a Kinibi service first installs the usage rights for its Kinibi trusted application (TA). This is done as follows:

a) **Program X** communicates over a **binder interface** with the **RootPA** application. This is an Android (Java)-internal operation, but the binder operations may require the correct SEAndroid policy to be allowed.

b) The **RootPA** communicates with **the daemon**. This communication is established over UNIX domain sockets, and is a privilege governed by SEAndroid policy.

c) **Program X** will initiate network communication with a remote Trusted Application Manager server, but parts of the protocol messages will be handled by a Kinibi TA. For this to happen, the **RootPA**, needs to **access the** `/dev/mobicore-user` device. Also, the **daemon** needs to access the `/dev/mobicore` device**.** Device access is done over ioctls, reads and write, in all cases.

d) The **RootPA** gets protocol results that need to be stored on flash / disk. This is achieved by **the daemon** reading and writing into a directory, typically `/data/misc/mcRegistry` At this point, the installation of the program rights has completed, and files have appeared in `/data/misc/mcRegistry`. For **Program X** to execute, a similar (but slightly different) variation of the communication takes place.

e) **Program X** opens a UNIX domain socket to the **daemon** for loading the TA into Kinibi.

f) The **daemon** again accesses the `/dev/mobicore` device to set up the TA for running. The **daemon** will again also consult `/data/misc/mcRegistry` for program rights. These rights do not differ from rights needed during provisioning

g) **Program X** accesses `/dev/mobicore-user` device to communicate with its TA

2.2.6.2 SEAndroid Policy Requirements

The Kinibi system does not today leverage SEAndroid policy for its own protection. Therefore we state no requirements for enforcement.
Instead, the requirements purely stem from accessibility, i.e. which communication channels need to be available for what kinds of system stakeholders in order for the Kinibi system to be fully working.

There likely is additional policy needed to get the system up to a functional state (e.g. to initialize / transition the daemon into its domain or to activate the character device that are so system-policy-dependent that they must be considered in the relevant policy only):

1) `/dev/mobicore-user` needs to be accessible from any Android application.
The SEAndroid class needed for communication is `chr-dev`.
Permissions can be seen from [1], obviously needed ones are `ioctl, read, write, open`

2) `/dev/mobicore-user` needs to be accessible from the mobicore daemon.
The SEAndroid class needed for communication is `chr-dev`.
Permissions can be seen from [1], obviously needed ones `are ioctl, read, write, open`

3) The user-space daemon (`/vendor/bin/mcDriverDaemon`) needs to be accessible over UNIX domain sockets from any Android application AND RootPA.
The relevant SEAndroid class is unix_stream_socket, and permissions needed for access are at least `connectto, read, write, create, sendto, recvfrom, ioctl, setopt, getopt, newconn`.

   The user-space daemon `(/vendor/bin/mcDriverDaemon)` needs to be able to open a server unix stream socket: in addition to the permissions above, at least name_bind, acceptfrom and listen are needed.

4) The user-space proxy (`/vendor/bin/trustonic_tee_proxy`) needs to be accessible over UNIX domain sockets from any Android application AND RootPA.
The relevant SEAndroid class is unix_stream_socket, and permissions needed for access are at least `connectto, read, write, create, sendto, recvfrom, ioctl, setopt, getopt, newconn`.

   The user-space proxy (`/vendor/bin/trustonic_tee_proxy`) needs to be able to open a server unix stream socket: in addition to the permissions above, at least name_bind, acceptfrom and listen are needed.

5) The directory /data/misc/mcRegistry needs to be readable and writable by the daemon. Relevant SEAndroid classes are dir and file. From a security perspective it is good (but not absolutely necessary) to protect that directory against access from other system or application components.

2.3 INTEGRATION IN LINUX

2.3.1 Integration in Device File System

The table below gives the directories in the Android file system where Kinibi components must be placed:

2.3.1.1 Linux OS file system:

Component	Name of the binary	Containing folder
Kinibi Daemon	`mcDriverDaemon`	`/usr/bin/`
Kinibi Proxy	`trustonic_tee_proxy`	`/usr/bin/`
Kinibi Client library	`libMcClient.so`	`/usr/lib/`
Kinibi registry interface	`libMcRegistry.so`	`/usr/lib/`

If the Kinibi Linux driver modules are not built within the Linux kernel tree you must place them in the following locations:

Component	Name of the binary	Containing folder
Kinibi user device	`mcDrvModule.ko`	`/usr/lib/modules/`

A couple of Trusted Applications have to be placed into the registry folder:

Component	Name of the binary	Containing folder
Content Management Trusted Application	`07010000…0000.tlbin` *(tlcm.axf)*	`/usr/share/trustonic_tee/registry/`

2.3.2 Permissions and Access Rights

To enable the correct operation of Kinibi and any Trusted Applications it is mandatory to ensure all Kinibi components have the correct permissions and access rights. The following section highlights the required settings.

- Kinibi Daemon (`mcDriverDaemon`):
  The Kinibi Daemon should have permissions `0755`.
- Kinibi libraries (`libMc*`):
  No specific permission are needed for libraries.

- Kinibi Linux driver:
  The Linux driver needs two device nodes in `/dev` for the two different access mechanisms: `/dev/mobicore` for administration purpose, and `/dev/mobicore-user` for client access. Permissions for these two nodes must be defined in the following way:

```
/dev/mobicore-user 0666 XXX:XXX
/dev/mobicore      0600 XXX:XXX
```

  Note: `/dev/mobicore-user` is only created when the TEE has started successfully, which is triggered by the first daemon's connection to `/dev/mobicore`.

2.3.3 Directories Requirements

There are two directories that must be created in the device file-system for Kinibi. These paths are used by Kinibi components at runtime.

2.3.3.1 mcRegistry directory

This directory is used by Kinibi to store non-permanent data that can be deleted with a factory reset or device wipe.

The mcRegistry directory stores content relating to Device Binding, including:

- TEE Authentication Token (also known as the AuthToken)
- TEE Authentication Token backup (also known as the AuthToken backup)

Note that the TEE Authentication Token backup is an exception, and it must be preserved across factory reset or device wipe

The mcRegistry directory also stores content relating to the OTA-management of third-party content, including:

- Kinibi Root Containers
- Service Provider Containers
- Service Provider TA Containers
- Service Provider TA binaries (used only for TAs using GP APIs)

Service Provider and System TAs making use of GP Trusted Storage APIs automatically store Secure Objects in sub-folders of the mcRegistry directory.

The mandatory path is:

```
/usr/share/trustonic_tee/registry/
```

Permissions should be as follows:

- User: `XXX`, can read/write/execute
- Group: `XXX`, can read/write/execute
- Others can read/execute

In UNIX speak, permissions `0775`, user/group `XXX:XXX`.

2.3.3.2 Persistent Trusted Applications and Secure Drivers

Trusted Applications and Secure Drivers can be stored anywhere in the device file system. However, Persistent Trusted Applications or Secure Driver binaries must be placed in a non-volatile partition of the file system in order to survive a factory reset or device wipe.

Trustonic System Trusted Applications like the Content Management Trusted Application (CMTL) are must be found in:

```
/usr/share/trustonic_tee/registry/
```

Permissions should be as follows:

- User: `XXX`, can read/write/execute
- Group: `XXX`, can read/execute
- Others can read/execute

In UNIX speak, permissions `0755`, user/group `XXX:XXX`.

Please note that there is no content generated here at runtime by Kinibi, the directory is only used to store persistent System Trusted Applications.

2.3.4 Starting up Kinibi

For automatically running the Kinibi daemon at start-up a systemd script is given tee-daemon.service:

```
[Unit]
Description=Trustonic TrustZone daemon

[Service]
Type=simple
ExecStart=/usr/bin/mcDriverDaemon -b -p /usr/share/trustonic_tee/registry/
Restart=always

[Install]
WantedBy=multi-user.target
```

Copy tee-daemon.service in `/etc/system/system` and launch this command:

```
systemctl enable tee-daemon.service
```

The daemon has to have system permissions for `user` and `group` otherwise the OEM is responsible to set the correct parameters for the service.

The OEM is also responsible for deciding how the system should behave in the event of an error, for example, if the device fails to correctly initialize Kinibi or if the Kinibi daemon crashes.

3 KINIBI SECURE WORLD COMPONENTS

Secure World components of Kinibi are available in the package directory `/SecureWorldIntegration`. It contains:

- The Kinibi binary (secure operating system running in Secure World).
- The Content Management trusted application (System trusted application responsible for Containers management).
- Secure World drivers in skeleton form that the SIP and OEM have to integrate.
  - RPMB driver
  - Trusted User Interface Driver
  - DRM Driver (t-play)
- The ARM Trusted Firmware patch (spd, optional , for ARMv8 only)
- The TEE image builder (t-base-kit)
- The Android Keymaster Trusted Applications
- The MobiConfig tool.

The Kinibi images are provided in binary (Debug or Release) under the *`SecureIntegration/t-base/bin/MobiCore/`* directory.

Please note that the `DEBUG` version is much slower than the `RELEASE` version as it prints all Kinibi traces to the Linux Kernel log (`dmesg`). You can always get traces from trusted applications if they are compiled as Debug).

The images are un-configured (`mobicore.raw.img`) and should be configured with a hash of a system trusted application public key before using them. For reference the package contains also images signed with Trustonic test keys (`.img`).

The Content Management trusted application is provided in binary form (`DEBUG` and `RELEASE`) under `SecureIntegration/tbase/bin/TlCm`. The trusted application is provided un-configured (`tlCm.axf`) and should be configured with the Kinibi KPH request signing key and signed with the System trusted application key pair.

The Keymaster, Keymaster1.0 and Gatekeeper trusted applications are provided in source code and in binary form (`DEBUG` and `RELEASE`) under `SecureIntegration/TlTeeKeymaster`, `SecureIntegration/TlTeeGatekeeper`. The trusted applications are provided un-signed (`tlTeeKeymaster.axf`, `tlTeeKeymasterM.axf` and `tlTeeGatekeeper.axf`) and should be signed with the System trusted application key pair. Otherwise the build.sh scripts can be used to build the TAs from source code.

3.1 VERIFYING THE KINIBI VERSION

Before integrating Kinibi on a device, please verify the version tag in the binary. This version tag is changed each time Kinibi is built, so it offers a way to distinguish between different Kinibi versions and also between engineering and commercial releases.

To check the Kinibi version, look for the string `"t-base-"` in the binary image:

```
$ strings SecureIntegration/t-base/Bin/MobiCore/*/Release/t-base.img |
grep t-base-
```

This displays several version tags because the Kinibi image is made of several components that each have their version string. All the version strings inside the same image are identical.

A Kinibi package includes both a release and a debug build, located in directories named `Release` and `Debug` respectively. To distinguish between a release and a debug build in isolation, search for strings that are only present in the debug build. The following command shows a way to distinguish the two variants:

```
$ strings SecureIntegration/t-base/Bin/MobiCore/*/Release/t-base.img |
grep "MTK: sigma0 size"
$ strings SecureIntegration/t-base/Bin/MobiCore/*/Debug/t-base.img |
grep "MTK: sigma0 size"
S0CBMTK: sigma0 size: code=0x%08x data/bss=0x%08x
```

To verify the running version during development and testing, call the function `tlApiGetMobiCoreVersion` from a Trusted Application or `mcGetMobiCoreVersion` from a Client Application as described in the Kinibi API Documentation. See also Section 7.1 Basic verification and version check.

3.2 KINIBI IN BOOT CHAIN

To maintain security it is important that the Kinibi image is started up in the boot flow as early as possible. Booting up with Kinibi is a critical task in the system design and it should be implemented in co-operation with the silicon vendor and Trustonic. There are several options for implementing Kinibi in the boot chain so the purpose of this chapter is to give a basic level of understanding as to what needs to be done.

- It is recommended to store Kinibi at a permanent location in the Boot partition or some other permanent storage as early in the bootchain there is often no access to a regular file system.
- The Kinibi image must be signed by the OEM or ODM.
- The Kinibi image should be loaded to secure memory (internal or DRAM) by a bootloader such as Uboot.
- After Kinibi is loaded into secure memory its signature must be verified.
- Before calling Kinibi, the caller must correctly set the boot configuration block which is used for exchanging predefined data with Kinibi.
- Calling Kinibi can usually be just a basic jump instruction.

Please note that as the initialization routine of Kinibi returns to the caller, the caller environment (bootloader/uboot) should be saved and restored for the time period execution is in TrustZone side.

If the Kinibi signature check or boot fails it is up to OEM or SOC vendor to decide whether or not the device can continue to boot.

3.2.1 IRAM support

Trustzone memory protection hides secure memory from Android. On-chip memory (internal memory, or IRAM) provides additional protection against probing attacks and should be used to store cryptographic keys and similar secrets. Kinibi offers support for IRAM in three ways.

The bootloader can load Kinibi to IRAM memory. Thereby the Kinibi binary code can be protected from probing.

When the bootloader passes an IRAM region to Kinibi via the boot parameters (see 3.5), Kinibi will allow loading Trusted Applications that are converted with –memtype=1 into this memory.

Depending on the Kinibi version, Kinibi uses parts of the IRAM region from the bootloader for its own data structures:

Kinibi version	IRAM usage
310B	`MTK thread control block and L1 tables (80 KB)` `Crypto driver data and BSS (40 KB)`
Earlier versions	`Integration specific`

3.3 TRUSTED APPLICATIONS AND DRIVERS

In a Kinibi environment, there are 3 types of secure components:

- System Trusted Applications,
- Service Provider Trusted Applications,
- Drivers.

System Trusted Applications and drivers are pre-integrated into the device before release. Conversely, Service Provider Trusted Applications are deployed at runtime, over the air, by a Trusted Application Manager (TAM). This document focuses on pre-integrated applications.

To ensure the security of these Trusted Applications or drivers they need to be signed and verified at runtime by Kinibi. To do this the hash of the System Trusted Application's public key is inserted into the Kinibi raw-image.

By convention, trusted applications are named as `UUID.tlbin` and drivers as `UUID.drbin`, where UUID is a 16-byte hex value generated according to RFC-4122.

Any UUID value can be used by an OEM however UUID's with the last 12 bytes all zero are reserved for Trustonic:

- *0000 0000 0000 0000 0000 0000 0000 0000*,
- *0000 0001 0000 0000 0000 0000 0000 0000*,
- … ,
- *FFFF FFFF 0000 0000 0000 0000 0000 0000*.

For more details on Trusted Application and driver development please consult the `Kinibi_Developer's_Guide.pdf`.

3.4 CONFIGURING AND SIGNING KINIBI COMPONENTS

The following section describes how to manage the different private and public keys to ensure the authenticity and integrity of the Kinibi binary, drivers, and System Trusted Applications.

It is mandatory for the OEM to go through all the steps in this section to customize the keys involved for the correct and secure operation of Kinibi.

3.4.1 Keys in Kinibi Environment

There are four keys involved in the Kinibi environment that are of special importance during the integration phase:

- `PrK.Vendor.Boot` and `PuK.Vendor.Boot`:
  A PKI key pair used during boot for verifying the integrity of the Kinibi image.
- `PrK.Vendor.TltSig` and `PuK.Vendor.TltSig`:
  A PKI key pair used for signing System Trusted Applications and drivers that are installed by default on the device.
- `PuK.Trustonic.Endorsement`
  Used by Trusted Applications to sign data that can be proven to originate from a genuine TEE and the right Trusted Application. This key is provided by Trustonic as part of the Kinibi-package.
- `Prk.Kph.Request` and `PuK.Kph.Request`:
  A PKI key pair used to in message exchanges with the Kinibi Content Management Trusted Application in a production environment. Further details of how this key is used is can be found in the documentation titled `Kinibi_Provisioning_Manual.pdf`.

OpenSSL is the recommended way to generate the key pairs described above:

```
$ openssl genrsa -3 -out VendorBoot.pem 2048
$ openssl rsa
    -in VendorBoot.pem
    -pubout
    -outform PEM
    -out VendorBoot_pub.pem
$ openssl genrsa -3 -out VendorTltSig.pem 2048
$ openssl rsa
    -in VendorTltSig.pem
    -pubout
    -outform PEM
    -out VendorTltSig_pub.pem
```

(PEM format is expected by Kinibi components).

3.4.2 Configuring Kinibi

3.4.2.1 Configuring the Kinibi Image

The first step in configuration is to inject a hash of the System Trusted Application signing public key (`PuK.Vendor.TltSig`) into the raw Kinibi image. The goal of this is to guarantee the origin of System Trusted Applications and drivers. Secondly the Trustonic endorsement key, located in the directory: *`SecureIntegration\t-base\bin\MobiCore\endorsementPubKey.pem`*, should be included to enable authenticity checks on Trusted Application data. To develop applications using this feature please refer to the `Kinibi Developer's Guide` for details about the

Endorsement API. An additional configuration step has to be performed on the Kinibi image to make the Trusted Storage driver use Rollback Protection. See chapter **Error! Reference source not found.** for more details.

The MobiConfig tool is provided to inject keys into a binary and can be found in the directory: `SecureIntegration/tools/MobiConfig`:

```
$ cd SecureIntegration/tools/MobiConfig
$ java -jar MobiConfig/Out/Bin/MobiConfig.jar
    -c
    -i mobicore.img.raw
    -o mobicore.img
    -k VendorTltSig_pub.pem
    -ek endorsementPubKey.pem
```

See also Appendix II for full list of MobiConfig options.

3.4.2.2 Signing the System Trusted Applications and Drivers

All System Trusted Applications and drivers must be signed with the signing key `PuK.Vendor.TltSig`.

The MobiConvert tool is provided to sign a Trusted Application and can be found in the directory: *`t-base-dev-kit/t-sdk/TlSdk/Bin/MobiConvert`*.

```
$ cd t-base-dev-kit/t-sdk/TlSdk/Bin/MobiConvert
$ java -jar MobiConvert.jar
    -b my_system_trustlet.axf.conf
    -servicetype 3
    -output 00010002000300040005000600070008.tlbin
    -k VendorTltSig.pem
```

3.4.2.3 Signing the Kinibi Image

Signing Kinibi with `VendorBoot.pem` has to be done by the OEM or ODM and a signature check of must be performed before starting up.

3.4.3 Configuring Content Management Trusted Application

The Content Management trusted application is the secure peer of the Normal World component RootPA. It is involved:

- In production, during device manufactory, to generate the Authentication Token.
- At runtime, once the device is deployed, for Content Management on the Secure side (components creation).

3.4.3.1 Configure the Content Management Trusted Application

The goal of this step is to inject the KPH request signing key (`PuK.Kph.Request`) into the Content Management Trusted Application so that Kinibi can authenticate the requests coming from the KPH.

Each key comes with a KID to identify it (in some use cases, several keys can be supported):

```
$ cd SecureIntegration/tools/MobiConfig
$ java -jar MobiConfig/Out/Bin/MobiConfig.jar
    -c
    -i tlCm.axf
    -o tlCm.axf.conf
    -k MobiConfig_GuD_KPH_public_key.pem
    --kid 0
```

3.4.3.2 Signing the Content Management Trusted Application

The Content Management Trusted Application is a System Trusted Application. Once configured with the KPH request signing key and its KID, it must be signed with the usual System Trusted Application Signing key (`PrK.Vendor.TltSig`):

```
$ cd t-base-dev-kit/t-sdk/TlSdk/Bin/MobiConvert
$ java -jar MobiConvert.jar
    -b tlCm.axf.conf
    -servicetype 3
    -output 07010000000000000000000000000000.tlbin
    -k VendorTltSig.pem
```

The signed Content Management Trusted Application must be installed on the device in the Registry and must survive a device factory reset or wipe.

3.4.4 Signing the Trusted User Interface Driver

The Trusted User Interface driver is also a new driver in Kinibi-300 that must be signed and installed on the device.

The number of threads of the TUI driver is dependent on the integration. It is **5** for the reference TUI integration on Arndale (DCI, IPC, Exception, touch and VSYNC thread). It may be more if integration specific code launches additional threads. For instance, TUI integration on Qualcomm based devices has **6** threads (DCI, IPC, Exception, touch and Blitter and SafeKeeper thread).

```
$ cd t-base-dev-kit/t-sdk/DrSdk/Out/Bin/MobiConvert
$ java -jar MobiConvert.jar
    -servicetype 1
    -numberofthreads 5 or 6
    -bin drTui.axf
    -output 070c00000000000000000000000000.drbin
    -d 257
    -memtype 2
    -flags 0
    -interfaceversion 1.0
    -keyfile VendorTltSig.pem
$ cp 070c00000000000000000000000000.drbin
070c00000000000000000000000000.tlbin
```

The signed driver must be installed on the device in the Registry and must survive a device factory reset or wipe.

3.4.5 Signing the DRM Driver

The DRM driver must be signed and installed on the device.

```
$ cd t-base-dev-kit/t-sdk/DrSdk/Out/Bin/MobiConvert
$ java -jar MobiConvert.jar
    -servicetype 1
    -numberofthreads 3
    -bin tlDrTplayDrm.axf
    -output 070b00000000000000000000000000000.drbin
    -d 1536
    -memtype 2
    -flags 0
    -interfaceversion 1.0
    -keyfile VendorTltSig.pem
$ cp 070b00000000000000000000000000000.drbin
070b00000000000000000000000000000.tlbin
```

The signed driver must be installed on the device in the Registry and must survive a device factory reset or wipe.

3.4.1 Signing the TEE Keymaster Trusted Application

The TEE KeyMaster Trusted Application is a new component in Kinibi-301 which must be signed and installed on the device.

```
$ cd t-base-dev-kit/t-sdk/TlSdk/Out/Bin/MobiConvert
$ java -jar MobiConvert.jar
    -b tlTeeKeymaster.axf
    -servicetype 3
    -output 07060000000000000000000000000000.tlbin
    -k VendorTltSig.pem
```

The TEE KeyMaster1.0 and Gatekeeper Trusted Applications are new components in Kinibi-302C and Kinibi-310B and must be signed and installed on the device.

```
$ cd t-base-dev-kit/t-sdk/TlSdk/Out/Bin/MobiConvert
java -jar MobiConvert.jar \
     -bin tlTeeKeymasterM.axf \
     -output 07060000000000000000000000004d.tlbin \
     -numberofinstances 2 \
     -servicetype 3 \
     -n 1 -memtype 2 -flags 8 -interfaceversion 77.0 \
     -initheapsize 2048 -maxheapsize 32768

java -jar MobiConvert.jar \
     -bin tlTeeGatekeeper.axf \
     -output 07061000000000000000000000000000.tlbin \
     -servicetype 3 \
     -n 1 -memtype 2 -flags 8 -interfaceversion 1.0 \
     -initheapsize 2048 -maxheapsize 32768
```

3.5 KINIBI BOOT PARAMETERS ON 32 BIT HARDWARE

The entity that starts up Kinibi, either the bootloader or Uboot, should use the boot configuration block as an interface to Kinibi. The configuration block can be customized according to needs but must transfer at least the following information to Kinibi.

Example of Kinibi boot arguments on a Uboot integration:

Register	Values
r0	0, Cold boot. 1, Wake up from sleep.
r1	Pointer to MCSysInfo block (physical address, MCSysInfo_ptr). Defined below.
r13 / sp	MC boot stack (physical address) / 20 words available
r14 / lr	Start address of NWd. Kinibi jumps to this NWD address after changing the NS bit.

For reference here is the definition of the Kinibi system information structure:

```
typedef struct {
    uint32_t            magic;       //
    uint32_t            version;     // 0x00010000
    uint32_t            length;      // 0x00000030 (bytes)
    uint32_t            flags;       // Reserved
    struct mem_info_t   dram_total;  // Total DRAM area
    struct mem_info_t   dram_sec;    // Secure DRAM area
    struct mem_info_t   sram_total;  // Total SRAM area
    struct mem_info_t   sram_sec;    // Secure sRAM area
} MCSysInfo_t, *MCSysInfo_ptr;

typedef struct {
    uint32_t       base; // physical address (32bits)
    uint32_t       size;
} mem_info_t;
```

3.6 KINIBI ON ARM64 HARDWARE

On this type of hardware platform, the entity that starts up Kinibi is the *Trusted Firmware* (reference version provided by ARM). To start a new integration, Trustonic will provide a reference version that includes an implementation of the *Secure Payload Dispatcher (spd)* in charge of the Kinibi component.

3.6.1 Kinibi Boot Parameters on 64 bit Hardware

For reference, here is the definition of the Kinibi system information structure:

```
typedef struct {
    uint32_t magic;         // magic value from information
    uint32_t length;        // size of struct in bytes.
    uint64_t version;       // Version of structure
    uint64_t dRamBase;      // NonSecure DRAM start address
    uint64_t dRamSize;      // NonSecure DRAM size
    uint64_t secDRamBase;   // Secure DRAM start address
    uint64_t secDRamSize;   // Secure DRAM size
    uint64_t secIRamBase;   // Secure IRAM base
    uint64_t secIRamSize;   // Secure IRam size
    uint64_t conf_mair_el3;// MAIR_EL3 for memory attributes sharing
    uint32_t RFU1;
    uint32_t MSMPteCount;   // Number of MMU entries for MSM
    uint64_t MSMBase;       // MMU entries for MSM
    uint64_t gic_distributor_base;  // GIC dist base address
    uint64_t gic_cpuinterface_base; // GIC CPU base address
    uint32_t gic_version;           // GIC version
    uint32_t total_number_spi;      // Total SPI number in the system
    uint32_t ssiq_number;           // interrupt used for TEE comm.
    uint32_t RFU2;
    uint64_t flags;
} bootCfg_t;
```

3.6.2 SMC from SWd to ATF monitor

3.6.2.1 SMC_TEE_FC_INPUT

Kinibi uses the fastcall input to read platform-specific properties from the monitor.

Register	Value	Explanation
R0	0xB2000005	
R1	DataId	Identifier for data to be read, see below for possible IDs
R2	Length	Requested data length in bytes

Data is returned as follows:

Register	Value	Explanation
R0	Status	Status code, 0=ok.
R1	Offset	Offset to MSM
R2	realLen	Real length of data in MSM.

DataId indicates read data as follows:

ID	Value	Explanation
1	HWIdentity	Chip identity information (SUID), total of 96 bits.
2	HWKey	Chip –specific random key value.
3	RNG	Read random number based on true random number generator.
4	Freq	Platform frequency in kHz, divisor for CNTPCT.
5	Firmware version	Get ATF version information string, official version.
6	Firmware binary version	Get ATF version information string, exact version like git revision.

The sample code in SecureIntegration/spd shows how to implement these callbacks.

To start a new integration, Trustonic will provide additional documentation.

3.7 FASTCALLS HOOK MECHANISM

A FastCall is a call to the ARM SMC instruction with some specific parameters that allow the execution of some routines in Kinibi without performing a complete context switch.

FastCalls are always executed in IRQ Mode with FIQ and IRQ masked and must therefore execute as little, carefully designed, code as possible.

The calling convention for the ARM SMC instruction for performing FastCalls is as follow:

FastCall Parameters (Input)

```
r0          FastCallID (always < 0)
r1 – r3     FC parameters depending on the FastCallID
```

Return Values (Output)

```
r0          FastCallID (always < 0) of the FC which has been executed
            (r0 from input data)
r1 - r3     status information / data (depending on FC)
            (r1 returns status "0" or error)
```

Kinibi already supports either generic or platform-specific internal FastCalls. They are mainly used for Kinibi initialization and common operations that have to be done by the Normal World but can only be performed in the Secure World.

FastCalls have the following limitations:

- They cannot call any TlApi or DrApi functions
- They may be executed concurrently on several CPUs
- They must not cause any exception. There is no means to recover in case an exception is triggered by a FastCall.

3.7.1 Handling FastCalls

Kinibi allows two Secure Drivers known as Firmware Drivers to register an additional FastCall handler that will be called for FastCall IDs that are not known to Kinibi. "*Firmware Driver*" is a naming convention for the first two drivers to install a FastCall handler during Kinibi runtime. It is intended to act as a system integration means, compared to other hardware peripherals' drivers. One Firmware Driver is intended to be developed by the Silicon Provider (SiP) to handle FastCalls related to the silicon platform and the second Firmware Driver is intended to be developed by the OEM to handle FastCalls related to the device.

As it must always be available as soon as the FastCall handler has been registered, the Firmware Driver **must** be marked as permanent in the driver flags defined in its `Makefile`:

```
DRIVER_FLAGS := 1 # 0: no flags; 1: permanent; […]
```

The Firmware Driver routine gets called whenever Kinibi kernel's FastCall handler receives an ID designated by ARM as a SIP or OEM fastcall ID:

- SiP fastcall ID > 0x81000000
- OEM fastcall ID > 0x83000000

By calling the `drApiInstallFc()` function, the Firmware Driver indicates which FastCall ID it is handling.

Regarding the information shared between the Firmware driver and the Fastcall handlers: the Fastcall handlers get Firmware driver memory mappings in the range of 0-2MB at the time the handler is installed. As there is absolutely no synchronization mechanism between the Firmware driver and the FastCall handlers once installed, it is mandatory for the Firmware driver to not unmap any of these mappings.

In addition, if new mappings are made by the Firmware driver after FastCall installation, they cannot be relied upon to be visible in Fastcall hook functions.

3.7.2 Additional Secure Driver APIs

The main header file for the Driver API extension is `DrApiFastCall.h`.

```
#include "DrApi/DrApiFastCall.h"
```

3.7.2.1 Types

3.7.2.1.1 FastCall Registers

```
typedef uint32_t *fastcall_registers_t;
```

Depending on the platform, a FastCall handler may have access to at least 4 registers (r0 to r3).

Each time an SMC is sent and caught by the Firmware driver, they are forwarded by Kinibi to the Firmware driver through the `fastcall_registers_t` table. The size of this table can be determined by field, `registers`, from FastCall context structure (`struct fcContext`)

3.7.2.1.2 FastCall Context

```
struct fcContext {

     /* Size of the context */
     uint32_t size;

     /*Callback to modify L1 MMU mapping */
     void *(*setL1Entry)(fcContext_t context,
                         uint32_t idx,
                         uint32_t entry);

    void *(*setL1Entry64)(struct fcContext *context,
                          uint32_t idx,
                          uint64_t entry);

     /* Number of registers available in FastCalls handler */
     uint32_t registers;

     void (*prepareIdenticalMapping)(struct fcContext *context,
                                     addr_t start,
                                     uint32_t length,
                                     uint32_t flags);   void
(*generateFcNotification)(struct fcContext *context);
};
```

The FastCall context structure, `fcContext_t`, is filled when the Firmware driver is initialized in Kinibi and forwarded as a parameter to the FastCall hook initialization function.

Note: This context is shared between FastCalls and all processors.

The `setL1Entry` callback can be used at runtime to map additional memory in the FastCall context:

- `context`: The context parameter to the fastcall hook.
- `idx`: The Index of the section in the table of L1 descriptors; this is highly system dependent. In the current Kinibi version values from 0 to 7 are valid.
- `entry`: The L1 descriptor that will be used for the mapping.

The value returned by the function `setL1Entry` is the virtual address of the mapped area (`NULL` in case of error).

However, these mappings, done in the FastCall context, will not be visible from the Firmware driver.

The `prepareIdenticalMapping` callback can be used at runtime to affectively make one to one mapping for a given memory area. One to one mapping is mapping where the virtual address is same as the physical address.

- `context`: The context parameter to the FastCall hook.
- `start`: The start address of the one to one mapping area.
- `length`: The size of the one to one mapping area.
- `flags`: Currently not in use.

The address range [`0x0000 0000 – 0x01ff ffff`] is today reserved only for Kinibi internal usage. In consequence, the range passed in for one to one mapping must not overlap this area as any mapping in the Kinibi range will be ignored.

The `generateFcNotification` callback can be used at runtime to generate a notification interrupt from the FastCall hook context to a Secure World driver (by default this feature is based on SGI 8).

Both of the above features are highly system dependent and may not be available in all Kinibi versions.

3.7.2.2 Specific FastCall Entry Points

3.7.2.2.1 FastCall Handler Initialization

```
typedef uint32_t (*fcInitHook)(
      struct fcContext *context
);
```

This entry point is called once when FastCall Hooking is enabled through a call to the `drApiInstallFc` driver API (Firmware driver initialization). It is executed in Secure SVC mode.

This function must **never** cause any exception.

Parameters

- `context`: FastCall context structure.

Returns

It must return 0 if initialization is successful otherwise, with any other value, the FastCall will not be allowed.

3.7.2.2.2 FastCall Handler

```
typedef uint32_t (*fcEntryHook)(
      fastcall_registers_t *regs,
      struct fcContext *context
);
```

This is the actual FastCall handler. It may be executed concurrently on several CPUs and is executed only in IRQ mode.

This function must **never** cause any exception.

Parameters

- `regs`: Normal World registers' values:
  - On entry, `regs[0]` to `regs[context->registers - 1]` contain input parameters. `regs[0]` is always the FastCall identifier.
  - On exit, `regs[0]` to `regs[3]` store output results.
    - `regs[0]` is always the FastCall identifier.
    - `regs[1]` can be used to store a return value.
    - `Regs[2]` free to use
    - `Regs[3]` free to use
    - Other registers **must not** be modified; the result of any modification is unpredictable.

    By convention, if the FastCall identifier is unknown the value `MC_FC_RET_ERR_INVALID` should be returned in r1.
- `context`: FastCall context structure.

Returns

A non zero value should be returned when a FastCall is handled internally and zero when the FastCall is not known.

3.7.2.3 Specific Firmware Driver APIs

3.7.2.3.1 drApiInstallFc

```
_DRAPI_EXTERN_C drApiResult_t drApiInstallFc(
      void *entryTable,
      uint32_t fastcallOwner)
```

Install the custom FastCall handler.

Parameters:

- `entryTable`: table of function pointers to FastCall Hooking entry points (see section 3.7.3, "Firmware Driver Structure").
  - `entryTable[0]` should point to a function of type `fcInitHook`
  - `entryTable[1]` should point to a function of type `fcEntryHook`
- `fastcallOwner`: define which FastCall IDs will be handled by the hook. Currently supported values :
  - `FASTCALL_OWNER_SIP`
  - `FASTCAL_OWNER_OEM`

Returns:

- `DRAPI_OK` if the FastCall handler has been correctly set
- `E_DRAPI_NOT_PERMITTED` if this function is called from a non-driver context
- `E_DRAPI_INVALID_PARAMETER` if `entryTable` does not point to code in the driver
- `E_DRAPI_CANNOT_INIT` if the driver has not been configured as permanent
- `DRAPI_ERROR_CREATE(E_DRAPI_CANNOT_INIT, E_MAPPED)` if another FastCall handler has already been installed

3.7.3 Firmware Driver Structure

3.7.3.1 FastCall Hook Initialization

Initialization of FastCall handling is done by the Firmware driver, it boils down to building a 2-entries table where:

- the first element is a pointer to the FastCall initialization function,
- the second one is a pointer to the actual FastCall handler.

```
void *entryVector[2] = { &fcInit, &_fcMain };

_DRAPI_ENTRY void drMain(
    const addr_t    dciBuffer,
    const uint32_t  dciBufferLen
){
      drApiResult_t ret = drApiInstallFc(entryVector);
      if (E_OK != ret)
      {
          drDbgPrintf("Initialization failed: %x\n",rv);
          ...
      }

      /* Start IPC handler */
      drIpchInit(dciBuffer, dciBufferLen);
}
```

3.7.3.2 Assembly Glue for the FastCall Handler

The FastCall handler, once installed, will be executed in the context of the Monitor and eventually on any CPUs available. While the main Monitor might have a decent stack, it might not be the case for the secondary monitors. For this reason, it's required to setup a new stack before entering in the FastCall Handler

The following piece of assembly code is an example of how to reserve specific stacks for each CPU on which the FastCall handler may run and use the correct one when it is executed (`_fcMain` is the function installed, it then calls the real FastCall Handler, `fcMain`):

```
      export _fcMain
      import fcMain

CORES_MAX   equ  48
STACK_SIZE  equ  256 * 4

      area stack, noinit, readwrite
fcStackBottom
      space CORES_MAX * STACK_SIZE
fcStackTop

      area text, code, readonly
      preserve8
      arm
_fcMain
      push { r6-r8, r12, lr}
      mov r12, sp

      ; get affinity level 0 core number in r1
      mrc    p15, 0, r7, c0, c0, 5    ; get mpidr
      ubfx   r6, r7, #0, #4           ; cpu id(1-3)
      ubfx   r7, r7, #8, #4           ; cluster id (8-11)

      ; set own core-specific stack
      ldr    r7, =fcStackTop
      mov    r8, #STACK_SIZE
      ; calculate stack-start for this core
      mul    r6, r6, r8
      ; by sp = top - (core_num * size_core)
      sub    r7, r7, r6
      mov    sp, r7

      ; save the old stack in the new stack
      push {r12}
      blx fcMain
      ; get the old stack back
      pop {r12}
      mov sp, r12
      ; restore the context from the old stack
      pop { r6-r8, r12, lr}
      bx lr

      end
```

3.7.3.3 FastCall Handler Example

The FastCall Handler is divided into two steps:

- The initialization function, called by Kinibi when the Firmware driver installs the FastCall hook:
- `uint32_t` fcInit(fcContext *context)

```
{
    /* Initialization code here…
     * Runs in Kernel context */

    /* optionally map things… */
    void     *virt     =     context->setL1Entry(context,     0,
fcMakeL1PTE(phys_addr));
    return 0;
}
```

- The FastCall Handler, called each time an SMC is not recognized by Kinibi Monitor:

```
uint32_t fcMain(
    fastcall_registers_t *regs,
    fcContext *context)
{
    uint32_t mpidr;
    uint32_t fastCallID = regs[0];

    switch(fastCallID){
        case FASTCALL_SOMETHING:
            doFCSomething(regs);
            return 1;        default:
            break;
    }
    return 0;
}
```

3.7.4 u-boot Integration Example

This section describes an example integration of the Firmware driver in u-boot. An advantage of this approach is that the FastCall hook is available for u-boot and for the very early boot sequence of the Normal World OS.

3.7.4.1 Entry Point in u-boot

The proposed entry point in the bootloader code:

```
    ulong mobi_drv_addr = 0x0;
    size_t mobi_drv_size = 0x0;

    /* Initialize the MobiCore runtime */
    if(mci_setup())
        return CMD_RET_USAGE;

    if (mc_load_driver(mobi_drv_addr, mobi_drv_size, tci, TCI_SIZE))
        printf("MobiCore Driver loading failed!\n");

    // Unmap the MobiCore runtime - otherwise Linux daemon will fail
    mci_unmap();
```

It is up to the bootloader developer to specify:

- `mobi_drv_address` – the address in memory where the secure driver blob has been loaded to memory
- `mobi_drv_size` – the size of the blob loaded to memory
- `tci` – a global buffer already allocated to send commands to the driver
- `TCI_SIZE` – the size of the tci buffer previously allocated

This guide assumes that the bootloader developer has a mechanism to load the secure driver from permanent storage to memory AND has done so before integrating. It is also assumed that if a TCI is required the bootloader will not release that particular memory for other uses

3.7.4.2 Kinibi Communication Setup

The function `mci_setup()` handles the setup of the communication mechanism between the bootloader and Kinibi

The code provided in our example patch is assumed complete and working:

```
static int mci_setup(void)
{
      struct fc_generic fc_init;
      int i;

      uint32_t mci_offset = ((uint32_t)mci) & PAGE_MASK;
      fc_init.cmd = MC_FC_INIT;
      // Set MCI as uncached
      fc_init.param[0] = (uint32_t)mci | 0x1U;
      fc_init.param[1] = (mci_offset << 16 ) | NQ_LENGTH;
      // mcp_offset = 0x118 mcp_length=0x90
      fc_init.param[2] = ((NQ_LENGTH + mci_offset) << 16) | MCP_LENGTH;
      _smc(&fc_init);
      printf("MobiCore INIT response = %x\n", fc_init.param[0]);
      if(fc_init.param[0]) {
            printf("MobiCore MCI init failed!\n");
            return -1;
      }

      // MCI is setup, do a nsiq to give RTM a chance to run
      for(i = 0; i < MC_MAX_SIQ; i++)  {
            uint32_t state, ext_info;
            mc_nsiq();
            mc_info(0, &state, &ext_info);
            // Check if initialized
            if(state == MC_STATUS_INITIALIZED) {
                  printf("MobiCore RTM has initialized!\n");
                  break;
            }
      }
      if (i == MC_MAX_SIQ) {
            printf("MobiCore RTM failed to initialize\n");
            return -1;
      }
      mcp = (mcpBuffer_ptr)((uint8_t*)mci + mci_offset + NQ_LENGTH);
      nq = (uint8_t*)mci + mci_offset;
```

```
	printf("MobiCore IDLE flag = %x\n", mcp->mcFlags.schedule);

	return 0;
}
```

The code above assumes the buffer `MCI` is already allocated in memory and has a size of `MCI_SIZE`:

```
#define MCI_SIZE  512
uint8_t mci[MCI_SIZE];
```

This buffer must be globally defined as it is used for communication throughout the code! In addition this code executes several SMC calls to give Kinibi time to initialize. It does also have a maximum number of calls(`MC_MAX_SIQ`) defined so if something goes wrong it can handle errors correctly.

3.7.4.3 Firmware Driver Loading

The function for driver loading below is assumed complete and working.

The return value of the function is 0 for success – driver has been loaded and has initialized correctly.

```
static int mc_load_driver(void *buf, size_t size, void *tci, size_t tci_size)
{
	int ret = 0;
	int i;
	// now we have the driver in memory, setup the MCP
	mclfHeaderV2_ptr header = buf;
	printf("MobiCore driver address %x, size = %u!\n", buf, size);
	mcp->mcpMessage.cmdOpen.cmdHeader.cmdId = MC_MCP_CMD_OPEN_SESSION;
	mcp->mcpMessage.cmdOpen.uuid = header->uuid;
	mcp->mcpMessage.cmdOpen.wsmTypeTci = WSM_CONTIGUOUS | WSM_WSM_UNCACHED;
	mcp->mcpMessage.cmdOpen.adrTciBuffer = ((uint32_t)tci) & ~(PAGE_MASK);
	mcp->mcpMessage.cmdOpen.ofsTciBuffer = ((uint32_t)tci) & PAGE_MASK;
	mcp->mcpMessage.cmdOpen.lenTciBuffer = tci_size;

	// check if load data is provided
	mcp->mcpMessage.cmdOpen.wsmTypeLoadData=WSM_CONTIGUOUS|
WSM_WSM_UNCACHED;
	mcp->mcpMessage.cmdOpen.adrLoadData = ((uint32_t)buf) & ~(PAGE_MASK);
	mcp->mcpMessage.cmdOpen.ofsLoadData = ((uint32_t)buf) & PAGE_MASK;
	mcp->mcpMessage.cmdOpen.lenLoadData = size;
	memcpy(&mcp->mcpMessage.cmdOpen.tlHeader,header, sizeof(mclfHeader_t));

	put_notification(0);
	for (i = 0; i < MC_MAX_SIQ; i++) {
		mc_nsiq();
		udelay(2000);
		if(get_notification() == 0) {
			printf("MobiCore RTM Notified back!\n");
			break;
		}
	}
	if (i == MC_MAX_SIQ) {
		printf("MobiCore RTM did not ack the open command!\n");
		ret = -1;
	}

	for (i = 0; i < MC_MAX_SIQ || mcp->mcFlags.schedule; i++) {
		mc_nsiq();
		break;
	}
	if (i == MC_MAX_SIQ) {
```

```
            printf("MobiCore is not yet IDLE - driver is probably
missbehaving!\n");
            return -1;
      }
      printf("MobiCore Driver loaded and RTM IDLE!\n");
      return ret;
}
```

Parameters

- `buf` – the address in memory where the secure driver blob has been loaded to memory
- `size` – the size of the blob loaded to memory
- `tci` – a global buffer already allocated to send commands to the driver
- `tci_size` – the size of the tci buffer previously allocated

As stated before please note that if the secure driver uses the TCI after initialization you **must** ensure the memory allocated for the TCI is not released to Android. Furthermore, the driver must not have long initialization times or endless loops. The code above has checks that the driver will not take too long to initialize but since there might not be any time source interrupts in the bootloader Kinibi might not relinquish control to the bootloader at all,

3.7.4.4 Data Deallocation

Data deallocation is very important in the bootflow. Without proper deallocation subsequent calls to Kinibi from the Android daemon will fail.

The code below is considered complete and working:

```
static int mci_unmap(void)
{
      int i;
      mcp->mcpMessage.cmdHeader.cmdId = MC_MCP_CMD_CLOSE_MCP;
      put_notification(0);

      mc_nsiq();

      for(i = 0; i < MC_MAX_SIQ; i++)  {
            uint32_t state, ext_info;
            mc_info(0, &state, &ext_info);
            mc_nsiq();
            // Check if initialized
            if(state != MC_STATUS_INITIALIZED) {
                  printf("MobiCore RTM has been uninitialized!\n");
                  break;
            }
            mc_nsiq();
      }
      if (i == MC_MAX_SIQ) {
            printf("MobiCore RTM failed to uninitialize\n");
            return -1;
      }
      return 0;
}
```

Error handling is important as any error returned by this code will result in an unusable Kinibi

system for Android

3.8 FIQ FORWARD MECHANISM

Trustonic TEE allows forwarding a FIQ (usually processed at EL1 level) to the ARM Trusted Firmware in order to process it with the maximum priviledged level (EL3).

Even if this mechanism allows virtually any processing to be done at EL3 level for a given interrupt, developer must keep in mind the following things:

- At the time the dedicated callback is reached at EL3 level (plat_tbase_forward_fiq(), more details in chapter FIQ Forward handling), **the FIQ is already acknowledged and terminated from a GIC perspective (GICC_EOIR has been written for this interrupt)**.
- Developer must NOT trigger a world-switch from this place.

As detailed in following chapters, using this feature requires some effort during integration phase in order to

1. configure the forward mechanism
2. implement the associated handler for forwarded FIQs.

3.8.1 FIQ Forward configuration

Configuring the FIQ forward mechanism can be achieved by calling the following function for each FIQ that has to be forwarded and processed at EL3 level.

```
void tbase_fiqforward_configure(uint32_t intrNo, uint32_t enable);
```

Parameters

```
intrNo:
      this data must be set to the interrupt ID to be configured
enable:
      TBASE_FLAG_SET will enable the FIQ forward mechanism for the
target interrupt
      TBASE_FLAG_CLEAR will disable the FIQ forward mechanism for the
target interrupt
```

Returned value

```
      Not applicable
```

Calling this function must be done during the Trustonic TEE boot phase.

In order to keep the SPD structure simple and agnostic of this process, another function has been added to plat_tbase.c file as a dedicated placeholder for FIQ forward configuration calls:

(note: plat_tbase.c is the common placeholder for platform abstraction of Trustonic TEE's SPD to the silicon platform)

```
void plat_tbase_fiqforward_init(void)
```

```
{
        printf( "Configuring TEE forwarded FIQs...\n");

        /* Watchdog FIQ configuration */
        tbase_fiqforward_configure( WDT_IRQ_BIT_ID,    /* interrupt id
*/
                                    TBASE_FLAG_SET    );    /*    enable
forward */

        /* Another forwarded FIQ, just for testing purpose */
        tbase_fiqforward_configure( 162,               /* interrupt id
*/
                                    TBASE_FLAG_SET    );    /*    enable
forward */
}
```

Previous implementation is an example on how to configure the mechanism to enable forwarding interrupts WDT_IRQ_BIT_ID and 162 to the ARM Trusted Firmware.

Default implementation will be empty, and it's up to the silicon partner to enable/disable FIQ forwards during the platform integration step by calling appropriately `tbase_fiqforward_configure()` from the `plat_tbase_fiqforward_init()` function.

3.8.2 FIQ Forward handling

At runtime, each time a FIQ is triggered, Trustonic TEE kernel will check if it has to be forwarded to the ARM Trusted firmware.

1. If IT IS NOT the case, it will be normally processed by the kernel and dispatched to the task waiting for it (if any).
2. If IT IS the case, the kernel will not dispatch this interrupt further to any tasks running in the TEE and will trigger a dedicated fastcall (hidden to the developer), finally calling the following function with the forwarded interrupt number as parameter :

```
uint32_t plat_tbase_forward_fiq(uint32_t fiqId)
```

Parameters

```
fiqId:
     this data holds the interrupt number of the forwarded FIQ.
```

Returned value

```
     PLAT_TBASE_INPUT_OK  if  the  appropriate  processing  was  done
successfully.
     PLAT_TBASE_INPUT_ERROR otherwise.
```

Please note the returned value is ignored by the Trustonic TEE kernel as of today, but could potentially be used in later versions.

As an example, for the same interrupts as in FIQ Forward Configuration chapter, the following code could be used:

```
uint32_t plat_tbase_forward_fiq(uint32_t fiqId)
{
        uint32_t Status = PLAT_TBASE_INPUT_OK;
```

```
        uint32_t linear_id = platform_get_core_pos(read_mpidr());

        /* Verbosity */
        printf( "core %d EL3 received forwarded FIQ %d from the TEE
!\n", (int)linear_id, (int) fiqId);

        /* Handle forwarded FIQ */
        switch (fiqId)
        {
        case WDT_IRQ_BIT_ID:
                /* Dump the platform… */
                platformDump();
                break;
        case 162:
                /* just a simple test */
                printf("%s: That's a test !\n",__func__);
                break;
        default:
                /* Unknown FIQ */
                printf("%s: FIQ %d was forwarded but no processing was
associated to it.\n",__func__,fiqId);
                Status = PLAT_TBASE_INPUT_ERROR;
                break;
        }


        return Status;
}
```

Default implementation will be empty and it's up to the silicon partner to implement the expected processing in **`plat_tbase_forward_fiq()`**, during the platform integration step.

4 TEE KEYMASTER AND GATEKEEPER

The Kinibi TEE KeyMaster feature provides support for all cryptographic algorithms required to be compliant with the Google Android KeyMaster implementation.

The product package contains the Keymaster implementation as required for Lollypop and Kitkat (see 4.1 Keymaster0) as well as the Keymaster and Gatekeeper implementation for Android Marshmallow and Nougat (see 50 Keymaster1). The same architectures is used for both versions, but the integration is closer to the default in the Marshmallow and Nougat versions.

4.1 KEYMASTER0 – LOLLYPOP AND KITKAT

The following algorithms are supported:

- RSA (both normal and CRT) up to 4096 bit key sizes. Raw RSA operations are also supported.
- DSA up to 3072 bit key sizes.
- ECDSA (curves P-192, P-224, P-256, P-384 and P-521).

And the following operations are supported for RSA, DSA and ECDSA

- Generate key data,
- Sign,
- Verify,
- Import key data ,
- Get public key.

These cryptographic features are available through two additional components, a Trusted Application Connector and a Trusted Application:

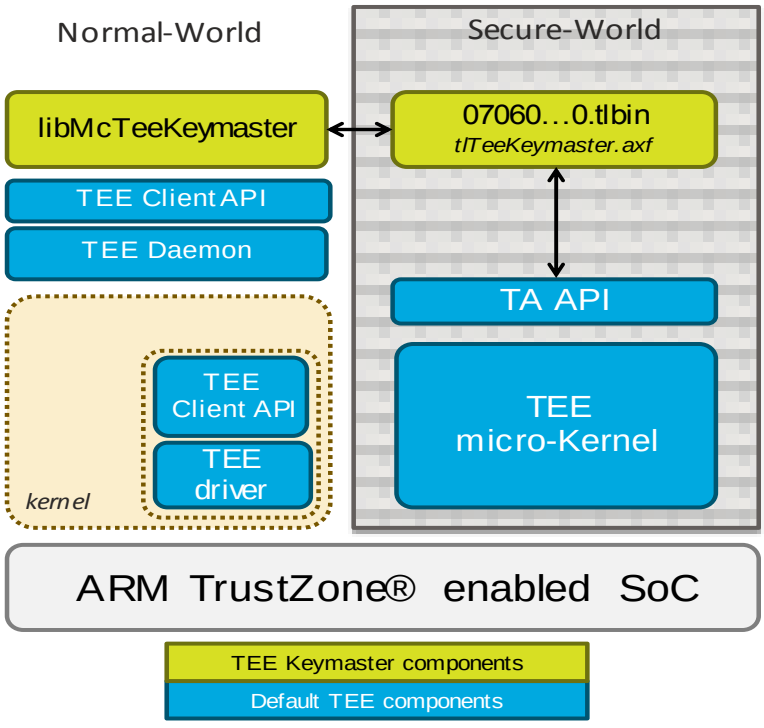


Figure 2: TEE Keymaster architecture

4.1.1 Normal World Connector and Integration

The Kinibi package contains the Normal World library implementation connecting Google's Android KeyMaster framework and the Kinibi secure OS.

This library, `libMcTeeKeymaster.so`, (which is also a Trusted Application Connector) is provided in source and binary forms. It can be found at:

```
AndroidIntegration/Bin/TlcTeeKeymaster/
```

The header file, `tlcTeeKeymaster_if.h`, defines the interfaces to communicate with Kinibi and is available at:

```
AndroidIntegration/Src/TlcTeeKeymaster/Locals/Code/public/tlcTeeKeymaster_if.h
```

It has to be installed in the Android libraries directory into the device file system under:

```
/system/lib/libMcTeeKeymaster.so
```

The HAL keystore module in the Linux kernel is not included in this library and is not part of the Kinibi product. However, as a generic statement, KeyMaster HAL module development would involve the following steps:

- Implement `HAL_MODULE_INFO_SYM` and assign the correct KeyMaster functions in `xxxxx_km_open` APIs. For example:

```
static struct hw_module_methods_t keystore_module_methods = {
    open: xxxxx_km_open,
};

struct    keystore_module    HAL_MODULE_INFO_SYM    __attribute__
((visibility ("default"))) = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: KEYSTORE_HARDWARE_MODULE_ID,
        name: "Keymaster XXXXX HAL",
        author: "YYYY ZZZZ",
        methods: &keystore_module_methods,
        dso: 0,
        reserved: {},
    },
};
```

- Then, in that particular file, implement the KeyMaster APIs (`generate_`, `import_`, `get_keypair_`, `sign_`, `verify_`) with calls to Kinibi TEE keymaster APIs detailed in interface file `tlcTeeKeymaster_if.h`.

› Finally, update `Android.mk` to generate the file `keystore.xxxx.so` (which should be placed on the device in the following directory: `/vendor/lib/hw/keystore.xxxx.so`

```
MOBICORE_PATH := hardware/yyyyzzzz/xxxx/mobicore

LOCAL_MODULE := keystore.xxxx
LOCAL_MODULE_PATH := $(TARGET_OUT_VENDOR_SHARED_LIBRARIES)/hw
LOCAL_SRC_FILES  :=  keymaster_mobicore.cpp  tlcTeeKeymaster_if.c
LOCAL_C_INCLUDES := \
      external/openssl/include \
      $(MOBICORE_PATH)/daemon/ClientLib/public \
      $(MOBICORE_PATH)/common/MobiCore/inc/
LOCAL_C_FLAGS     =     -fvisibility=hidden     -Wall     -Werror
LOCAL_SHARED_LIBRARIES    :=    libcrypto    liblog    libMcClient
LOCAL_MODULE_TAGS     :=     optional     LOCAL_MODULE_CLASS     :=
SHARED_LIBRARIES

include $(BUILD_SHARED_LIBRARY)
```

4.1.2 Secure World Component

The TEE KeyMaster Trusted Application is the link between the Normal World connector and the Secure World implementation. This component is mainly an interface on top of Kinibi, where the different cryptographic algorithms are implemented.

The source and the binary of the Trusted Application are available in the Kinibi package under:

```
/SecureIntegration/TlTeeKeymaster/Out/Release/tlTeeKeymaster.axf
```

Once signed, this Trusted Application will be named `07060…0.tlbin`.

4.2 KEYMASTER1 AND GATEKEEPER – MARSHMALLOW AND NOUGAT

4.2.1 Normal World Connector and Integration

The Kinibi package contains the Normal World library implementation connecting Google's Android Keymaster1.0 and Gatekeeper framework and the Kinibi secure OS. The Android tree build will build the libraries and place them into /vendor/lib/hw and /vendor/lib64/hw respectively. The keystore daemon will try to open the /vendor/lib/hw/keystore.$DEVICE.so and thereby use the Kinibi Keymaster1.0 implementation. Similarly, the gatekeeper daemon will try to open the /vendor/lib/hw/gatekeeper.$DEVICE.so and use the Kinibi Gatekeeper implementation.

4.2.2 Secure World Component

The TEE KeyMaster1.0 and TEE Gatekeeper Trusted Applications implement the Keymaster and Gatekeeper functionality in the security of the Secure World and with the help of Kinibi secure OS.

The source and the binary of these Trusted Applications are available in the Kinibi package under:

```
/SecureIntegration/TlTeeKeymaster/Out/Release/tlTeeKeymasterM.axf
/SecureIntegration/TlTeeGatekeeper/Out/Release/tlTeeGatekeeper.axf
```

Once signed, these Trusted Applications will be named `07060…004D.tlbin` and `070610…0.tlbin` respectively.

Specific t-base-302 integration work on Gatekeeper Trusted Application side:

In current implementation, full Gatekeeper logic in localized secure. This Trusted Application is not persistent and is finally continuously loaded/unloaded by Android framework. The problem is to keep in memory the status of the failure records…

Kinibi-310 / Kinibi-311

No real issue in this configuration, the Trusted Application will store these failure records directly in the GP Secure Filesystem. The Gatekeeper integration is relatively straightforward and should not require customization.

t-base-302

This configuration is more complicated as GP SFS is not present in this TEE version. The solution implemented in Trustonic proposal is to use an additional Secure Worldd driver `DrAndroid`, kept always loaded in memory and finally used as storage in RAM.

The usage of this `DrAndroid` is controlled by the C Macro `__TRANSIENT_FAILURE_RECORD_STORAGE__`, which one is enabled from the TA makefile with the option `USE_TRANSIENT_FAILURE_RECORD_STORAGE`. Looking at the TA source code in a Kinibi package, the real call from Gatekeeper TA is done here:

```
SecureIntegration/TlTeeGatekeeper/Locals/Code/src/failure_record.c
```

function `write_transient_failure_record()` and `read_transient_failure_record()`.

4.2.3 Android Verified Boot and bootloader integration

This feature is a way for the platform bootloader to share a seed information with the TEE KeyMaster 1.0 Trusted Application, allowing it to differentiate its master key.

Integration work on the bootloader side:

The bootloader will be responsible to generate a hash (32-bytes buffer) over the bootloader pubkey and the Android lockState.

Then send this 32-bytes hash buffer generated to the TEE with 4 calls to the SMC instructions:

```
smc (SET_ANDROID_DEVICE_STATE, wordIndex, word_x, word_x+1);

SET_ANDROID_ DEVICE_STATE defined to 0xFF000039 (for legacy reason it
        is set to -203 on ARMv7 platform).
wordIndex would be 0, 1, 2, or 3, any others will be rejected.
word_x and word_x+1, the two 4-byte words the bootloader wants to
        store in the TEE at index wordIndex.
```

For security reason, once a word has been written, it cannot be re-written (to prevent later modification from the NWd OS).

Integration work on Android KeyMaster1.0 Trusted Application side:

Warning: By default Trustonic reference Trusted Application is using a hardcoded stub.

It is expected for the integrator to replace this stub by its own implementation or Trustonic VerifiedBoot solution.

The abstraction layer is implemented in the following file in KeyMaster1.0 Trusted Application:

```
SecureIntegration/TlTeeKeymaster/Locals/Code/M/util/tlkm_device.c
```

Once the platform bootloader code has been modified to set the device-state buffer, the integrator will have to update the KeyMaster1.0 Trusted Application and unset Macro `USE_HARDCODED_DEVICE_STATE_STUB` used in this file.

In case of Trustonic VerifiedBoot solution selected, getting the device state buffer is already implemented and based on the following interface:

```
ret    =    TEE_GetPropertyAsBinaryBlock(TEE_PROPSET_TEE_IMPLEMENTATION,
"com.trustonic.android.deviceState", &buff[0], &size);

buff, needs to be allocated by the callers.
size, param, need to be passed by the caller, but implementation can
        modify it depending of buffer size returned.
ret, can be TEE_SUCCESS or TEE_ERROR_BAD_STATE, in case the device
        state has not been correctly done (4 index of the hash buffer
        called)
```

5 DRM INTEGRATION

Kinibi supports a Secure Driver for DRM which allows Trusted Applications to process DRM content through a set of standard APIs. The Secure Driver once implemented must provide a means for the DRM Trusted Applications to decrypt encrypted content data and store the resulting data in protected regions of memory. The Secure Driver is responsible for verifying that the entire process is kept secure, this includes:

- Verifying the entire address range for output data is protected.
- Verifying integrity of decoder firmware (if required, according to design)
- Managing multiple sessions if multiple DRM sessions are supported concurrently.

5.1 HIGH LEVEL FLOW

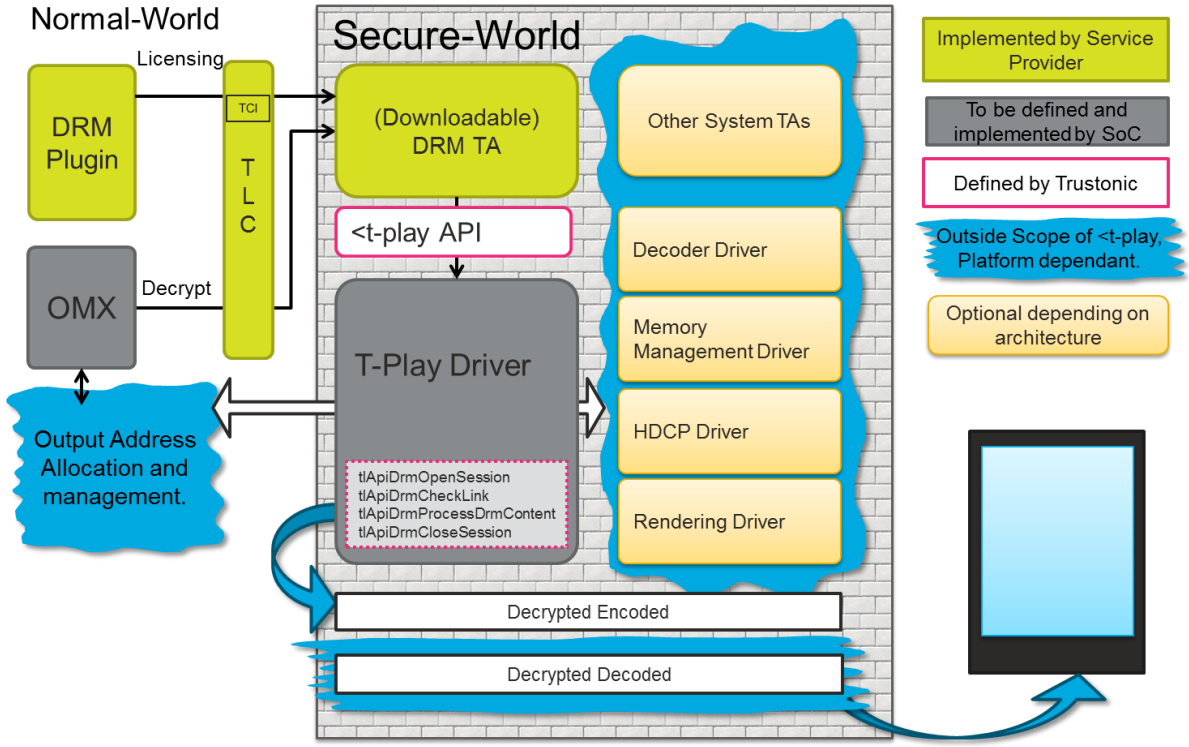


Figure 3: DRM components overview

The following is an example of one possible high level sequence :

1. The DRM Secure Driver will be authenticated by the secure OS.
2. During runtime the application is passed a DRM protected Stream.
3. The DRM Plugin handles the license acquisition with the DRM Trusted Application.
4. ACodec::allocateBuffersOnPort is called by the OMX component and a platform dependant mechanism is initiated to identify the output buffer to be used for the decrypt.
5. The value returned from allocateBuffersOnPort is set as the output reference and shall be passed unmodified to the DRM driver when the decryption is being processed.
6. The NW Client will allocate a region of world shared memory and place the encrypted content in it.

7. The Trusted Application calls the DRM Driver driver `openSession` function. If multiple sessions are going to be supported the session handle can be passed if known, and null if a new session is being opened.
8. The driver must initialize any hardware that will be required for example the crypto accelerator, decoder hardware, and raise the firewalls to protect any pre-defined memory regions.
9. If the encryption algorithm is not supported by the driver the Trusted Application will decrypt the content itself (software based) and call the DRM driver's `processDrmContent` function passing the clear data and the function's 'processmode' parameter set to 'PLAIN'. The DRM Secure Driver will need to copy this data into the predefined protected buffer and signal the media framework.
10. If the encryption algorithm is supported, the `processDrmContent` function will be called with processMode set to decrypt, the output address will be resolved according to format of the parameter passed via output and the data decrypted to this address. The media framework will then be signaled to know the encoded clear data is ready.
11. The decryption of data will continue until it has been fully successfully decrypted at which point the Trusted Application will call `closeSession` and finalize the operation by cleaning buffers and disabling firewalls.
12. The handling of the decoding and rendering is outside the scope of this document and of the DRM APIs.

5.2 T-PLAY ASSUMPTIONS

1. Mixed Input Data

It is assumed that if the buffer received by the TA contains mixed clear and encrypted segments, all data must be passed to the driver to be managed. The same restriction applies as above, in that the output address is not known by the TA. This can be done using the APIs defined to first copy the clear data to the output buffer and then decrypt the rest to overwrite the relevant segments.

5.3 DRIVERS OVERVIEW

5.3.1 Framework Support

The APIs defined by this document are provided in a stub form. If the DRM functionality is required it is necessary to implement a driver for the platform according to the hardware specifications. If the functions are not implemented the error `DRIVER_NOT_IMPLEMENTED` will be returned by default to any calls to the driver.

5.3.2 TLC and TA Driver Access

A trusted driver can only have one normal world client, therefore if we wish to communicate with the driver from both a TLC and a Trusted Application we must open the session using the TLC that will use the DCI directly with the driver. The session must then remain open which will allow the TA to access the driver.

It is advised that the session be opened and the driver be loaded at boot time, and then it can remain loaded and be accessible when required. It would also be recommended to implement the TLC as a kernel module, to restrict the access rights on who can use the driver APIs.

5.3.3 Driver-Client Access Control

Access control is not required in the driver as no assets are available to the Trusted Application in clear. If an unauthorized Trusted Application attempts to access the driver it cannot gain access to protected content. Please see Section 4. Security and Evaluation Considerations.

5.3.4 Threads

Below, the threads required in the implementation of the drivers are described, for more information on their usage please see [DrDrvGuide].

5.3.4.1 Exception Handler Thread

This is the main thread and runs with higher priority than IPC handler thread. Its main responsibility is to handle exceptions caused by the IPC handler thread. The exception handler is implemented in 'drTplayExcHandler.c'. When the IPC handler thread is started in drTplayIpcHandler.c (see `drIpchInit()`), the exception handler thread is registered as local exception handler. When the IPC handler thread causes an exception, the Kinibi kernel informs the exception handler. Then the IPC handler thread is restarted if exception is segmentation fault. If it is caused by, for example, undefined instruction, etc, the DRM Driver shuts down.

5.3.4.2 IPC Handler Thread

This is the second thread and it handles IPC messages sent by Trusted Applications. The IPC handler thread runs with lower priority than the exception handler thread and it is implemented in 'drTplayIpcHandler.c' . When it receives IPC messages from Trusted Applications, it checks function id, processes incoming requests accordingly and responds to Trusted Applications with relevant status code. For various operations such as AES encrypt/decrypt and data copy requests, the driver should maps Trusted Application's address space to access Trusted Application data. '`drApiAddrTranslateAndCheck()`' API call can be used for this purpose.

5.3.4.3 DCI Handler Thread

This is the third required thread and it intercepts IPC messages sent by Trusted Application Connectors. The DCI handler thread runs with lower priority than the exception handler thread but a higher priority than the IPC thread and it is implemented in 'drTplayDciHandler.c' . When it receives IPC messages from the normal world, it should parse the incoming data, and process incoming requests accordingly, finally responding to Trusted Application Connector with relevant status code.

5.3.5 Protected Buffers

The address at which the protected buffers lie must be within a predefined range which must be retrieved by the driver at runtime. This can either be passed as a parameter from the normal world, hardcoded within the driver or passed from the kernel through a Kinibi interface. The chosen method is platform dependent. For more information on the protected buffers and security concerns see section 4 of this document.

5.4 DRM DRIVER PROTOCOL

The DRM agent Trusted Application will call the DRM tlApi.

The driver will call drApiMapClientAndParams which returns a marshaling parameter which is given in more detail in the following sections according to the driver and the command being executed.

DRM Driver

```
/**
 * Function IDs
 */
typedef enum {
FID_DR_OPEN_SESSION 1
FID_DR_CLOSE_SESSION 2
FID_DR_PROCESS_DRM_CONTENT 3
FID_DR_CHECK_LINK 4
} Sec_FuncID_t;
```

Table 1: Driver Command IDs

DRM Secure Driver UUID

The DRM Secure Driver UUID is :

070b-0000-0000-0000-0000-0000-0000-0000

Secure Playback driver ID

The DRM Secure Driver ID is : 1536

Marshalling Parameters Structure

```
/**
 * Union of marshaling parameters. */
/* If adding any function, add the marshaling structure here
 */
typedef struct {
    uint32_t     functionId;  /* Function identifier. */
    union {
        uint8_t                 *returned_sHandle;
        uint8_t                 sHandle_to_close;
        tlDrmApiDrmContent_t    drmContent;
        tlDrmApiLink_t          link;
        int32_t                 retVal;      /* Return value */
    } payload;
} tplayMarshalingParam_t, *tplayMarshalingParam_ptr;
```

Commands

The DRM driver supports the following commands:

0x00000001 (FID_DR_OPEN_SESSION)

0x00000002 (FID_DR_CLOSE_SESSION)

0x00000003 (FID_DR_PROCESS_DRM_CONTENT)

`0x00000004 (FID_DR_CHECK_LINK)`

5.4.1.1 FID_DR_OPEN_SESSION

Command ID

0x00000001

Effect

This command is used to set hardware and context configurations according to what will be required depending on the content that will be decrypted. Buffers and structures can be organized and initialized with values if required.

The firewalls must also be enabled by this command.

Error Code

- `TLAPI_DRM_OK` if operation was successful.
- `E_TLAPI_DRM_INTERNAL` general error in case of crypto problem
- `E_TLAPI_DRM_MAP` in case of error mapping memory to driver.
- `E_TLAPI_DRM_PERMISSION_DENIED` in case of rights access related issue
- `E_TLAPI_DRM_SESSION_NOT_AVAILABLE` in case the driver is busy and cannot open a session.
- `E_TLAPI_DRM_DRIVER_NOT_IMPLEMENTED` in case the function is not implemented.

5.4.1.2 FID_DR_CLOSE_SESSION

Command ID

0x00000002

Effect

The DRM Agent sends this command to the DRM Secure Driver to disable the firewalls and carry out any necessary cleanup required. The driver does not free any memory, it merely changes the property and raises a flag to indicate the firewall is enabled.

Any buffers used by the secure driver should be reset so that their content is not readable after the firewalls have been disabled.

Error Code

- `TLAPI_DRM_OK` if operation was successful.
- `E_TLAPI_DRM_INTERNAL` in case of failure.
- `E_TLAPI_DRM_DRIVER_NOT_IMPLEMENTED` in case the function is not implemented.

5.4.1.3 FID_DR_PROCESS_DRM_CONTENT

Command ID

0x00000003

Structure passed in Marshaling Parameter

```
typedef struct {
```

```
    uint8_t                         sHandle,
    TL_DRM_DecryptContext           decryptCtx,
    uint8_t                        *input,
    TL_DRM_InputSegmentDescriptor   inputDesc,
    uint16_t                        processMode,
    uint8_t                        *output
} tlDrmApiDrmContent_t, *tlDrmApiDrmContent_ptr;
```

Effect

The DRM Agent sends this command to the DRM Secure Driver to either send decrypted content to the protected buffer, or to pass encrypted content for the driver to decrypt into a protected buffer so as to be processed by the media framework.

The function takes an offset as parameter within the InputSegmentDescriptor structure that is used to calculate the exact location in the output buffer to place the clear data. The ouput buffer must be known to the driver at this point. This can be done by a number of ways depending on the implementation, for example the protected region may always be static and the address can be hardcoded within the driver, the kernel can pass the address dynamically via a Kinibi interface, or the values could be stored in a secure registry and read out only when required.

The driver must ensure that the data to be decrypted falls within the limits of the firewalled region.

Error Code

- `TLAPI_DRM_OK` if operation was successful.
- `E_TLAPI_DRM_INVALID_PARAMS` incorrect parameters in input.
- `E_TLAPI_DRM_INTERNAL` general Error in case of crypto problem
- `E_TLAPI_DRM_MAP` in case of error mapping memory to driver.
- `E_TLAPI_DRM_PERMISSION_DENIED` in case of rights access related issue
- `E_TLAPI_DRM_REGION_NOT_SECURE` if the memory for output is not protected
- `E_TLAPI_DRM_ALGORITHM_NOT_SUPPORTED` in case the algorithm is not supported.
- `E_TLAPI_DRM_DRIVER_NOT_IMPLEMENTED` in case the function is not implemented.

5.4.1.4 FID_DR_CHECK_LINK

Command ID

0x00000004

Effect

The DRM Agent sends this command to the DRM Secure Driver to check the external link information like HDCPv1, HDCPv2, AirPlay, and DTCP.

Error Code

- `TLAPI_DRM_OK` if operation was successful.
- `E_TLAPI_DRM_INTERNAL` in case of failure.
- `E_TLAPI_DRM_DRIVER_NOT_IMPLEMENTED` in case the function is not implemented.

5.5 SECURITY AND EVALUATION CONSIDERATIONS

5.5.1 Video Buffer Protection

Trustzone technology allows for two methods of protecting buffers, we will use the naming conventions as follows:

1. *Protected Memory* : The memory region is protected by an access control that is based on the Bus ID.
2. *Secure Memory :* The security bit of the memory region is enabled.

Depending on the implementation there will be up to three buffers to protect, a buffer to contain the decrypted encoded data, one for decrypted decoded and another for the display. An implementation may choose to use the same buffer for multiple uses if desired.

The memory can be protected by one of the following methods:

1. *A Trusted Normal World Component.*
   If we receive notification from a normal world component that the memory is protected it is imperative that the trust has already been established with that component.

2. *Bus master filtering.*
   In this case we enable the security at boot time and filter the access to the protected region according to Bus ID and thus can give access to only particular hardware such as the VPU.

3. *Using TZASC at runtime.*
   In this case the secure bit is enabled directly from within the driver.

In each of these cases there must be a check to ensure that the range of protected/secure memory is large enough to contain completely the input data so as not to leak any sensitive data to unprotected region.

5.5.2 Checking of Pointers

All pointers passed to the driver functions must be verified to ensure a bad address is not being used.

5.5.3 Input to Crypto Hardware

It must be verified that all sensitive inputs being passed to the crypto hardware (if used) are coming from secure memory.

5.5.4 Integrity of System Components

It is recommended that the video firmware is authenticated when loaded, and that it is loaded during the openSession command. This will ensure that the firmware is correct for each decrypt and avoids an attack where the firmware could be replaced after bootup.

The authenticity of any component can be verified with a hash/signature check, but it must be ensured that it is done using a public key stored in secure memory on the device.

5.5.5 Trusted Application Isolation

In order to protect the secrets between different DRM schemes sharing the same drivers Kinibi implements an isolation between trusted applications. Each trusted application executes in its own memory space and any persistant secure objects stored by the Trusted Application are only accessible by the Trusted Application thus protecting sensitive key data.

If multiple Trusted Application's require access to the Secure Driver then some form of session management or resource permissions must be implemented so as to avoid any denial of service.

5.5.6 Debug Attack

It is assumed that in any platform for commercial release that JTAG or similar debug functionality is disabled.

5.5.7 Reset Buffers

It is imperative that the memory used to store buffers, keys and any other assets during any part of the secure processes are correctly reset after use.

6 TRUSTED USER INTERFACE INTEGRATION

Kinibi comes with components and templates for integrating the Trusted User Interface (TUI) feature. The following diagram shows the Trusted User Interface architecture.

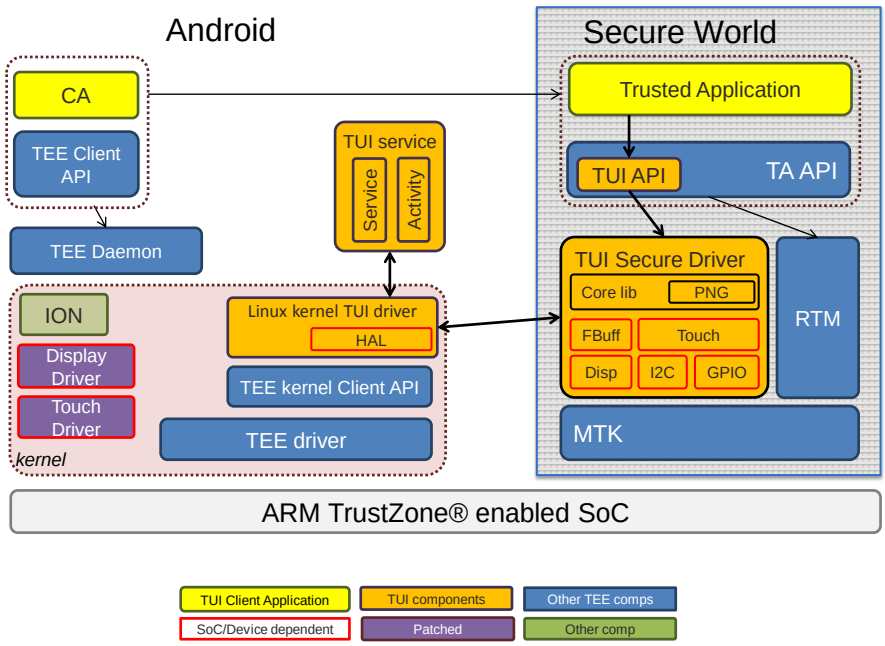


Figure 4: TUI components overview.

- The TUI application developer uses a TUI API as an extension of the TA API. This extension is hardware independent.
- The TUI secure driver in TZ Secure World is the core of the TUI implementation. It fully handles UI hardware within TUI session. It consists of a generic core part and a hardware abstraction layer (HAL) to be ported to the actual hardware. It is also internally linked with a TUI kernel driver running in the normal world.
- The TUI kernel driver is a proxy that links the TUI secure driver to the UI Linux drivers and a TUI Android service. It also contains a HAL to be ported to the actual kernel. The UI Linux drivers – typically touch, i2c and GPIO drivers for input, and display drivers – should be disabled and must not access the hardware within TUI session. They may need to be patched for this.
- Some system events must cancel an ongoing TUI session there is one. The TUI Service running in Android is designed to notice them and trigger the cancellation of the TUI session.
  The TUI service is started at boot time.
  Within TUI sessions the TUI service starts a TUI activity. The `android:keepScreenOn` attribute is set to true in TUI activity manifest and disables the Android screen timeout within TUI sessions.

Because TUI highly depends on the hardware of the device the TUI template cannot be considered as out-of-the-box product. The integrator must complete the porting for its own device.

6.1 SECURITY CONSIDERATIONS

The integrator must remember that the UI resources handled by the Secure-World (both hardware and software) manage sensitive information. This sensitive information must not be disclosed to the Normal-World during or after the TUI session. As part of ensuring this security requirement, Kinibi TUI has been designed to have exclusive access to UI resources within TUI session.

6.1.1 Framebuffer

Securing the display consists of securing the framebuffers, and optionally the display controller.

In case the framebuffers are allocated from secure world memory, the secure bootloader is responsible for setting up the TZASC and the framebuffers should not appear in any normal world mapping. The porting effort is limited to obtaining the physical address from the secure bootloader.

In case the framebuffers are allocated from normal world memory, they must be wiped before closing the TUI session. The TUI secure driver does it at every regular closing of the session. Furthermore, in case of a warm reset, the TUI secure driver may not have wiped the previous content of the buffers, therefore the secure bootloader must wipe the secure framebuffers from the previous boot before any normal world component is executed.

In case the framebuffers are allocated at fixed address the secure bootloader can just wipe it unconditionally. But in case it is allocated at variable address this address must be stored by the secure world in a location that is not lost after a warm reset. For this reason, it is recommended to locate the secure framebuffers at a fixed address.

If the framebuffers are allocated from the normal world, it is the responsibility of the secure integrator code to properly protect the buffer for secure-only access, for instance via TZASC. When the TUI driver receives the pointer and size of the framebuffer from the normal world, it passes it to a HAL function tuiHalFBProtect which should do the following:

- Check that the designated region consists entirely of normal world memory. If not, return a failure status.
- Change the TZASC configuration to make the buffer locked for secure access only.

These two steps must be done atomically. Failing to do it atomically may lead to race conditions if the framebuffers are shared with other secure services, like DRM.

6.1.2 Input devices

The input devices generally consist of an external touchscreen device linked to the main chip by an i2c interface and an additional GPIO as interrupt line.

The i2c bus is here critical because input data will transit on this bus. It must be fully protected during the TUI session. It means particularly that if the i2c bus is shared with other devices these other devices may not work properly during the TUI session.

The GPIO interrupt may be shared with many other devices in an interrupt decoding chain, making it very difficult to secure. The GPIO interrupt line is less critical as it only reveals that something has happened in input device. However it is sensitive because it may disclose the timing of key or button presses. It should be protected and managed by the secure world during TUI session.

The external touchscreen is critical too because it contains the last input events. It must be fully managed by the secure world during the TUI session. Furthermore particular attention must be paid to the sequence of operations when closing the TUI session. The touchscreen device should be reset before the closing completes, to ensure that the normal world cannot obtain any data on the last input events during the session. From a higher-level perspective, the input device should not be in use when closing the TUI session, to avoid disclosing the last action by the user. As the sequence of

operations to reset the device is hardware-specific, it needs to be customized by the hardware integrator. The following operations need to be performed:

1. Wait until the device state indicates that no finger is pressed.

2. Reset the input device so that it is subsequently impossible to read any information about past touch events.

6.2 TUI SECURE DRIVER

The TUI Secure Driver is provided as part of the Secure-World Integration components. It contains a part in source code – the HAL – and a part in binary – the core library. The integrator must complete the development of the HAL according to its platform and must recompile and sign the TUI Secure Driver image.

The following section give what needs to be taken care of in the Secure Driver.

6.2.1 Memory requirement

Double buffering:

The TUI driver requires three times the size of a full screen buffer as secure buffer.

This memory requirement is depending on both the resolution of the screen and the driver implementation.

- To reach the expected security level we are not considering any overlay or partial display. The secure display must be exclusive and full screen. The actual framebuffer size is then directly calculated from the display resolution.

  The current TUI driver implementation uses two framebuffers for rendering.

- The TUI core part needs a working buffer to uncompress images given by the TA. As images size can be up to full screen the working buffer is the same size as the framebuffer.

In case the secure buffers are allocated from Secure World memory (see Security Considerations) they must be reserved at boot time. Their addresses and size are recovered independently and successively using the function `tuiHalFBOpen`.

In case the secure buffers are allocated from Normal World memory (see Security Considerations) they must be contiguous. Their address and size are recovered at TUI session opening from the TUI kernel driver.

Drawing raw buffers of pixels:

The TUI driver allows Trusted Applications to draw raw buffers of pixels. They are allocated from TA memory space which is included in the whole TEE memory. To take benefit of drawing raw buffers the integrator should increase the permanent secure memory allocated for the whole TEE.

The recommended additional memory is about the same size as one frame buffer.

6.2.2 Lifecycle

The following entry point must be implemented by the integrator:

- `tuiHalGetVersion`: called at TUI driver loading to recover the version of the HAL software. If it does not match the version of the core library the initialization will fail.
- `tuiHalBoardInit`: called at TUI driver loading to give an initialization opportunity to the HAL software.

- `tuiHalHandleNwdMessage`: called when the TUI Secure Driver receives a message from the HAL of the NWd TUI kernel driver. The message contains one uint32_t of payload, whose meaning and interpretation is up to the integrator. If the TUI SWd HAL is not supposed to get any message from the TUI NWd HAL, then this function should do nothing.

The following callback is implemented by the core driver:

- `drTuiCoreCancelSessionRequest`: called at any time by the HAL to cancel a running TUI session if any.

6.2.3 Secure display

The entry points described in this section must be implemented by the integrator.

The following entry point is called at TUI secure driver loading:

- `tuiHalFBOpenV2`:
  - It is called at least three times, one for the working buffer, two for the framebuffers.
  - It must return the display metrics. Note that it includes the screen orientation: the integrator is responsible for returning the natural orientation of the device. It will not be changed by the TUI core driver.
  - It may return the physical address and size of the buffer in case the buffer is allocated from the Secure World memory. It only returns the size of the buffer in case it is allocated from Normal World memory.

The following entry points are called at TUI session opening:

- `tuiHalDisplayMapController`: setup MMU for the display system.
- `tuiHalDisplayProtectController`: setup TZPC for the display system.
- `tuiHalDisplayInitialize64`: initialize the display system for the Secure World.
- `tuiHalFBProtectV2`: setup TZASC for the framebuffer in case it is not taken from the Secure World memory. If you implement this function, please see guidelines from the “Security considerations” section.

The following entry points are called within TUI session:

- `tuiHalFBImageBltV2`: copy an image to a framebuffer. It may be a partial update of the framebuffer. The input format of raw image at this point is always the same: 32-bit per pixel (RGBA, where R is LSB). This function must support clipping.
- `tuiHalFBFillRect`: draw a rectangle filled with a given color to the destination framebuffer.
- `tuiHalFBPost`: post the framebufer and wait until it is displayed.
- `tuiHalFBCopyArea`: copy a rectangle area of the source framebuffer to a place in the destination framebuffer. This function must support clipping.

The following entry points are called at TUI session closing:

- `tuiHalFBUnProtectV2`: release TZASC for the framebuffer in case it is not taken from the Secure World memory. If you implement this function, please see guidelines from the “Security considerations” section.
- `tuiHalDisplayUnmapController`: release MMU for the display system.
- `tuiHalDisplayUnprotectController`: release TZPC for the display system.
- `tuiHalDisplayUninitialize`: release the display system.

The following callbacks are implemented by the core driver as a reference software implementation. They can be called by the integrator if no hardware implementation is available:

- `drTuiFBBlit`: draw a clipped rectangle area of an image to the destination FrameBuffer.
- `drTuiFBCopyArea`: copy a rectangle area of the source FrameBuffer to a place in the destination FrameBuffer.
- `drTuiFBFillRect`: draw a rectangle filled with a given color to the destination FrameBuffer.

6.2.4 Secure input

The TUI driver handles a particular thread fully dedicated to the secure input. This thread is referred as touch thread.

The touch thread has to be created and started by the HAL when opening the TUI session and killed by the core driver when closing the TUI session.

The following entry point must be implemented by the integrator:

- `tuiHalTouchOpen`: called from the core driver main thread at session opening. It must manage the input device during TUI session and particularly implement the touch thread. The synchronization of main and touch thread can be done using some callbacks. This function is responsible for the touch device initialization and should not return before it is done.
- `tuiHalTouchGetInfoV2`: called from the core driver main thread at session opening after `tuiHalTouchOpen`. The integrator must give the touch resolution and the desired size of touch event queue.
- `tuiHalTouchClose`: called from core driver main thread at TUI session closing. It should kill the touch thread and release the touch hardware.
  Before returning this function must particularly wait until the device state indicates that no finger is pressed and reset the input device so that it is subsequently impossible to read any information about past touch events.

The following callbacks are implemented by the core driver and must be called by the integrator to synchronize the touch thread:

- `drTouchReportV2`: called from the touch thread to signal a touch event to the core driver. The reported events are queued in the core driver until the TA reads them. In case of an overflow the oldest events in the queue will be overwritten. The size of the touch event queue should be tuned at integration time using the touch interactive tests.
- `drTouchLock`: called from the touch thread to signal to the core driver that the touch hardware is being used and it must not be interrupted or stopped.
- `drTouchUnlock`: called from the touch thread to signal to the core driver that the touch hardware is not being used anymore. It also means that the i2c interrupt must not be attached anymore to the touch thread when calling this.

  The integrator must take care not to call this function while the touch device state indicates that a finger is pressed.

  After calling this the touch thread may be killed and the touch device reset.

6.2.5 Building the TUI secure driver

The TUI core library `SecureIntegration/tui/DrTui/Out/Bin/Release/drTuiCore.a` implements the platform-independent part of the TUI driver. It is provided as binary only.

The integrator must complete the source part of the TUI secure driver and link it against the core library to generate the TUI driver.

In case the TUI driver UUID has to be changed it must be updated in `SecureIntegration/tui/DrTui/Locals/Code/public/dciTui.h` and `SecureIntegration/tui/DrTui/Locals/Code/driver.mk`.

For any new device the integrator must create a new `makefile.<device>.mk` in `SecureIntegration/tui/DrTui/Locals/Code` and update the source files and options for his particular device. The driver should be built using a new makefile variable `<DEVICE>`:

```
$ cd SecureIntegration/tui/DrTui
$ DEVICE=<DEVICE> ./Locals/Build/build.sh
```

6.2.6 Integrating the TUI secure driver

The TUI secure driver must be installed into the mcRegistry makefilepath, which is located in `/vendor/app/mcRegistry`

```
$ adb push SecureIntegration/t-
base/bin/DrTui/<PLATFORM>/<MODE>/<driver>.tlbin /vendor/app/mcRegistry
```

6.3 TUI KERNEL COMPONENTS

6.3.1 TUI kernel driver

The TUI kernel driver is a proxy between the TUI secure driver and the Normal world components – TUI service and Linux drivers.

The entry points described in this section must be implemented by the integrator.

The following entry point is called at TUI kernel driver loading:

- `uint32_t hal_tui_init(void)`
  This function is called when the TUI kernel driver is initialized, either at boot time, if built statically in the kernel, or when the kernel is dynamically loaded if built as a dynamic kernel module. This function may be used by the integrator, for instance, to get a memory pool that will be used to allocate the secure framebuffer and work buffer for TUI sessions. The function must return 0 on success, or non-zero on error. If the function returns an error, the module initialization will fail.
- `uint32_t hal_tui_exit(void)`
  This function is called when the TUI kernel driver exits. It is called when the TUI kernel driver is unloaded, if built dynamically, and never called if built-in into the kernel. It can be used to free any resources allocated by `hal_tui_init()`.

The following entry points are called at TUI session opening:

- ```
  uint32_t hal_tui_alloc(
                tuiAllocBuffer_t  *allocbuffer,
                size_t             allocsize,
                uint32_t           number
  )
  ```
  This function is called when the module receives a CMD_TUI_SW_OPEN_SESSION message from the secure driver. The function must allocate '`number`' buffer(s) of physically contiguous memory, where the length of each buffer is at least '`allocsize`' bytes. The physical address of each buffer must be stored in the array of structure '`allocbuffer`' which is provided as arguments. Physical address of the first buffer must be put in `allocate[0].pa`, the second one on `allocbuffer[1].pa`, and so on.
  The function must return 0 on success, non-zero on error.
  ```

For integrations where the framebuffer is not allocated by the Normal World, this function should do nothing and return success (zero).

- `uint32_t hal_tui_free(void)`
  This function is called at the end of the TUI session, when the TUI module receives the CMD_TUI_SW_CLOSE_SESSION message. The function should free the buffers allocated by `hal_tui_alloc(…)`.
- `uint32_t hal_tui_deactivate(void)`
  This function should stop the Normal World display and, if necessary, Normal World input. It is called when a TUI session is opening, before the Secure World takes control of display and input.

  It must return 0 on success, non-zero otherwise.
- `uint32_t hal_tui_activate(void)`
  This function should enable Normal World display and, if necessary, Normal World input. It is called after a TUI session, after the Secure World has released the display and input.
  It must return 0 on success, non-zero otherwise.
- `void hal_tui_post_start(struct tlc_tui_response_t *rsp)`
  This function is called after framebuffer allocation, when allocation is done using the Android Native Window API. If the HAL does not use Android Native Window API for framebuffer allocation, this function is not called.

The following functions are used for communication with the TUI Secure Driver HAL:

- ```
  uint32_t hal_tui_process_cmd(
        struct tui_hal_cmd_t *cmd,
        struct tui_hal_rsp_t *rsp)
  ```
  This function is called when a message is received from the TUI Secure Driver HAL. It receives as input the parameter '`cmd`', which contains the message payload and whose meaning is up to the HAL. The output parameter '`rsp`' must be filled with the response to the message to be forwarded to the TUI Secure Driver HAL. Its interpretation is up to the HAL.
- `uint32_t hal_tui_notif(void)`
  this function is called when the TUI Secure Driver HAL notifies the the TUI Kernel driver HAL. If the HAL is not expecting any notification from the TUI Secure Driver HAL, this function should do nothing.

The communication between TUI kernel module and TUI secure driver has several channels:

- Commands from the driver:
  - `CMD_TUI_SW_OPEN_SESSION`: TUI session is opening.
    - The TUI service must start watching the events that may cancel the TUI session.
    - In case the secure buffers are allocated from Normal World, they must be allocated now. The secure buffers must be contiguous and the total requested size is received as a parameter of this command.
    - The Linux drivers should stop using the UI hardware.
  - `CMD_TUI_SW_CLOSE_SESSION`:
    - The TUI service must stop watching the events that may cancel the TUI session.
    - The Linux drivers can use the UI hardware again.
    - In case the secure buffers are allocated in normal world they may be freed.
  - `CMD_TUI_SW_HAL`:
    - A message coming from the Secure HAL is received.
    - Actions to be done are up to the HAL. The callback `hal_tui_process_cmd` is called with the payload of the message and the response to be filled. The meaning and interpretation of the both the response and the response is up to the HAL.
  ```

- Responses from the normal world:
  - Each command from the secure driver must be acknowledged with a response from TUI kernel module. In particular the response to the CMD_TUI_SW_OPEN_SESSION command must contain the physical address of the secure buffer in case it is allocated from normal world memory.
- Notifications to the driver:
  - NOT_TUI_CANCEL_EVENT: notify the secure driver that an event that should cancel the TUI session was triggered in the TUI service.

6.3.2 Patching Linux drivers

The Linux drivers involved in UI – typically touch, i2c and GPIO drivers for input, and display drivers – must not access the hardware within TUI session. If not, security exceptions may be raised. The Linux drivers should be patched to avoid this and keep a safe behavior.

6.3.3 Building the kernel

This section is an addition of section 2.1.2 Kinibi Kernel-space Components.

It is assumed that input and video drivers have been patched according to the previous section.

The Kinibi TUI module can be found in `AndroidIntegration/Src/gud/TlcTui`

To enable automatic building inside the kernel follow these steps:

- Update Linux kernel configuration file `linux/drivers/gud/Kconfig` with following line (must be inserted at the end):

```
config TRUSTONIC_TRUSTED_UI
    tristate "Trustonic Trusted UI"
    depends on TRUSTONIC_TEE
    ---help---
      Enable Trustonic Trusted User Interface

config TRUSTONIC_TRUSTED_UI_FB_BLANK
    bool "Trustonic Trusted UI with fb_blank"
    depends on TRUSTONIC_TRUSTED_UI
    ---help---
    Blank the framebuffer before starting a TUI session
```

- And add the Kinibi TUI module to the Linux kernel build file `linux/drivers/gud/Makefile`:

```
obj-$(CONFIG_TRUSTONIC_TRUSTED_UI) += TlcTui/
```

- Then, run `make menuconfig` and in the *Device Drivers* page of the configuration menu, please select:
  - *Trustonic TEE Driver*
  - *Trustonic Trusted UI*
  - *Trustonic Trusted UI with fb_blank*

```
<*> Trustonic TEE Driver
[ ]   Trustonic TEE uses LPAE
[ ]   Trustonic TEE driver debug mode
<*>   Trustonic Trusted UI
```

```
<*>   Trustonic Trusted UI with fb_blank
```

› Finally, run `make` again and the Trustonic Trusted UI kernel components will be included in the kernel image.

6.3.4 Integrating the TUI kernel driver

The TUI kernel driver has one device for user access that need to be accessed by the TUI service. Access permissions for this device are defined by *udev* following way:

```
/dev/t-base-tui 0666 system:system
```

6.4 TUI ANDROID COMPONENTS

6.4.1 Customizing the TUI Service

Some system events must cancel an ongoing TUI session there is one. The TUI Android service is designed to notice them and trigger the cancellation of the TUI session.

The list of events can be customized in `com/trustonic/tuiservice/TuiService.java`:

› Add dynamic registration of intent in OnCreate()
› Add filtering of event in onReceive() of broadcastReceiver

Add required permission for intent reception if any in `AndroidManifest.xml`.

6.4.2 Integrating the TUI service

The usual way to build Kinibi TUI Android component is from a complete Android source tree:

Simply copy the contents of the folder `AndroidIntegration/Src/mobicore/TuiService/Locals/Code/` to an Android source tree. You should have the following organization:

```
Android root/
    External/
      mobicore/
        MobiCoreDriverLib/
        ProvisioningLib
        RootPA/
        TuiService
```

Add the following lines at the end of the file `External/mobicore/Android.mk`:

```
# Include the TUI Service
include $(MOBICORE_PROJECT_PATH)/TuiService/Android.mk
```

All Kinibi components – including TUI service – will be automatically built within the next build of full Android source tree.

Else usual Android command can be used to build only TUI component is:

```
$ mmm external/mobicore/Tuiservice/
```

6.4.3 SEAndroid configuration for TUI

Additionaly to the SE linux requirements for Kinibi some rules must be setup to allow TUI:

1) `/dev/t-base-tui` needs to be accessible from the TUI service.
   The SEAndroid class needed for communication is `chr-dev`.

Permissions can be seen from [1], obviously needed ones are `ioctl, read, write, open`

2) `/dev/t-base-tui` needs to be accessible from the user-space daemon (`vendor/bin/mcDriverDaemon`).
The SEAndroid class needed for communication is `chr-dev`.
Permissions can be seen from [1], obviously needed ones `are ioctl, read, write, open`

3) The user-space daemon (`/vendor/bin/mcDriverDaemon`) needs to be able to start the TUI service by calling the application manager (`am start com.trustonic.tuiservice/.TuiService`).
The user space daemon (`/vendor/bin/mcDriverDaemon`) must be added to a SEAndroid domain that is granted the execute permission for `dalvikcache_data_file`.

6.5 TUI FLOW CHARTS

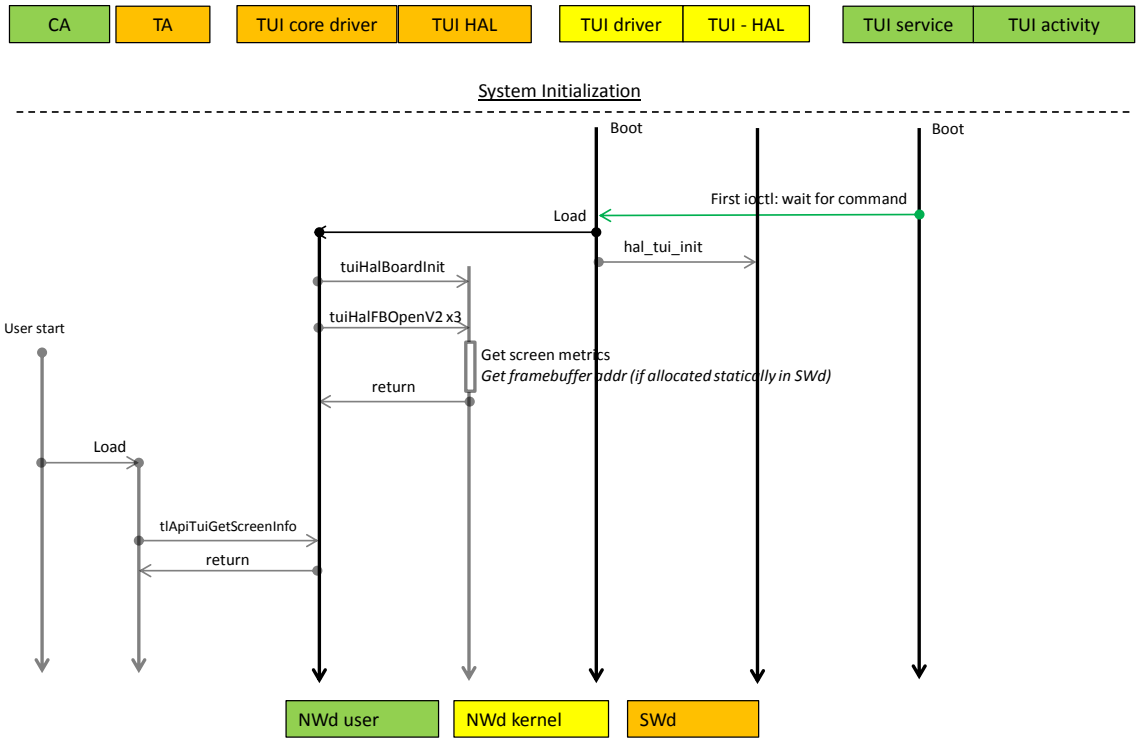


Figure 5: TUI flow chart: initialization.

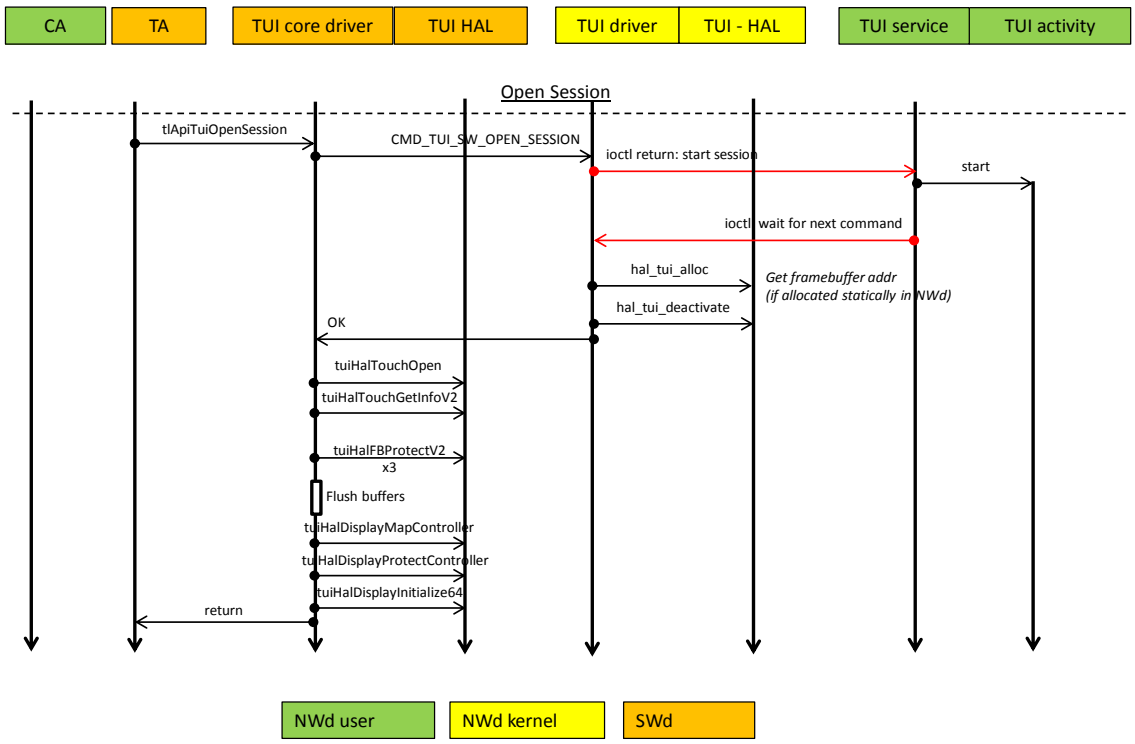


Figure 6: TUI flow chart: session opening.

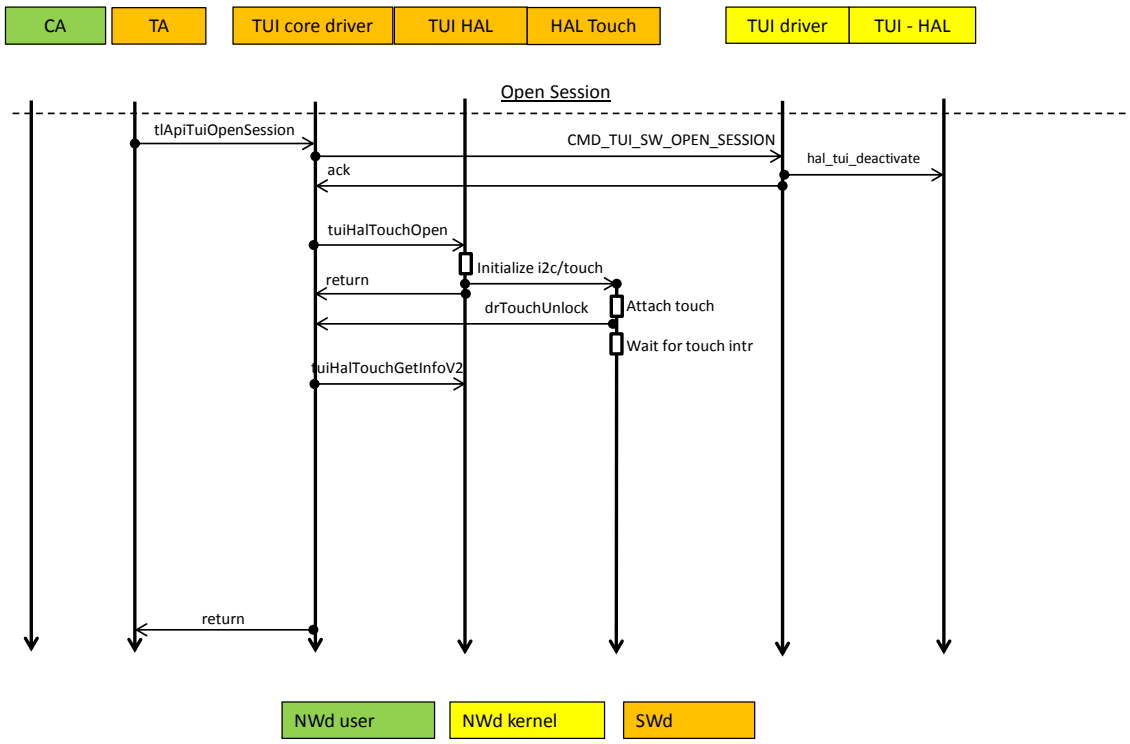


Figure 7: TUI flow chart: touch opening.

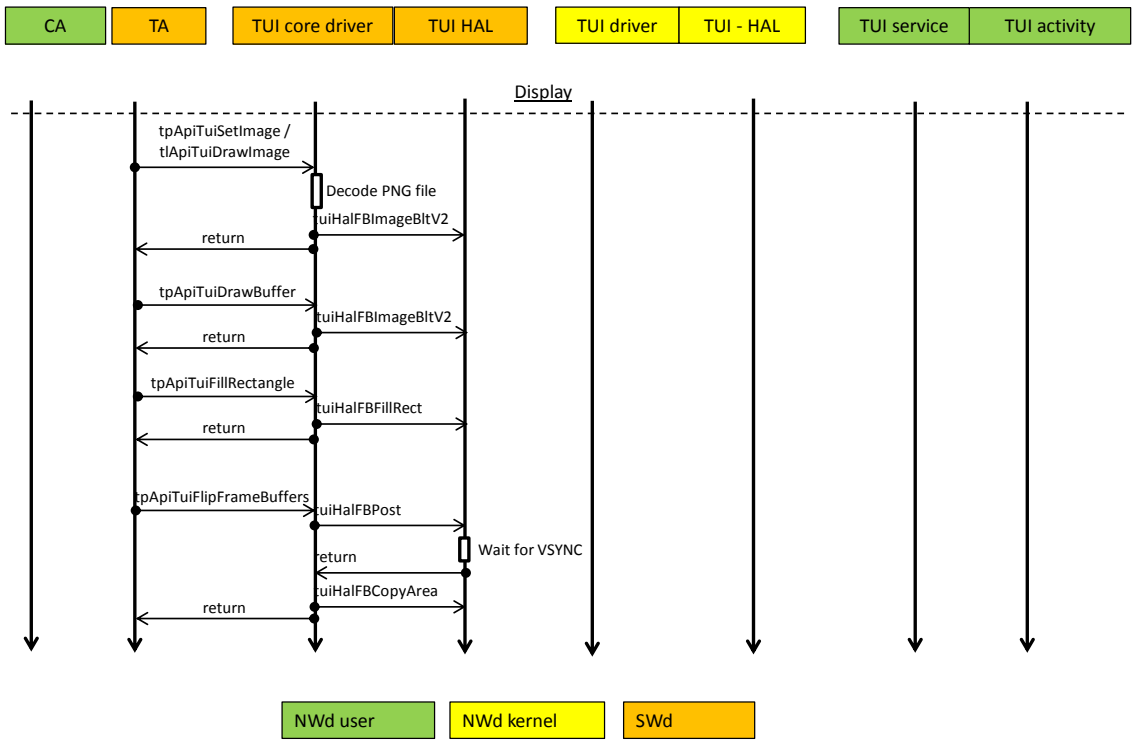


Figure 8: TUI flow chart: displaying.

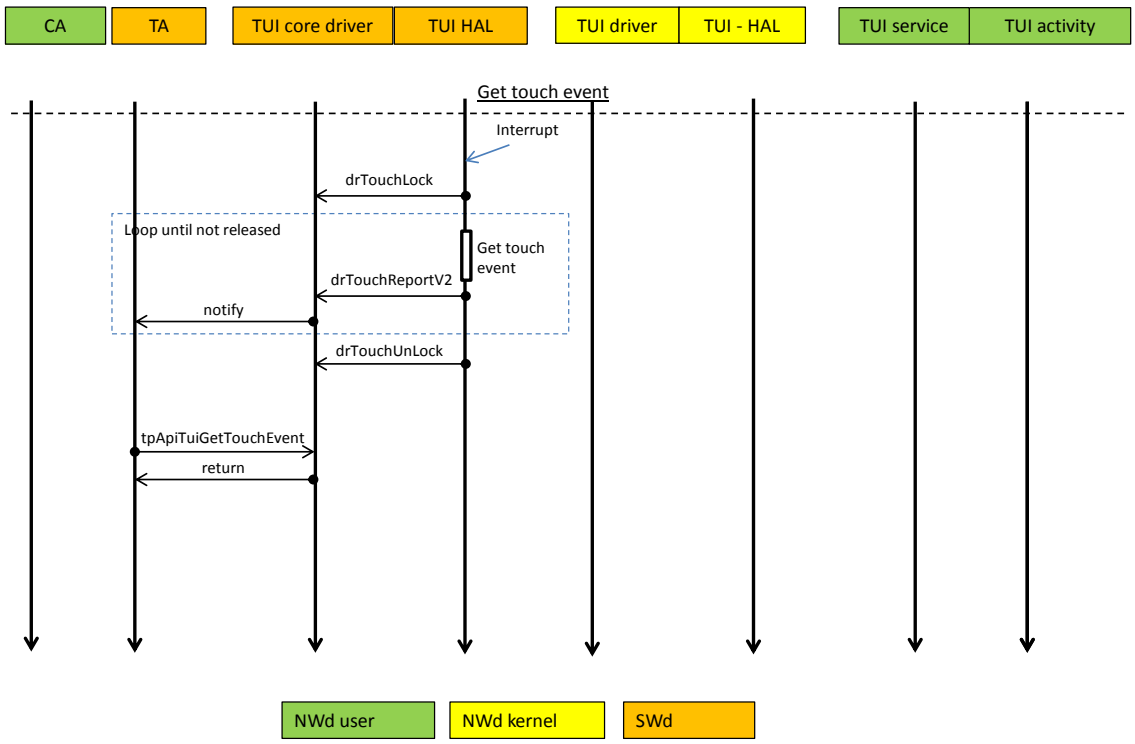


Figure 9: TUI flow chart: getting touch event.

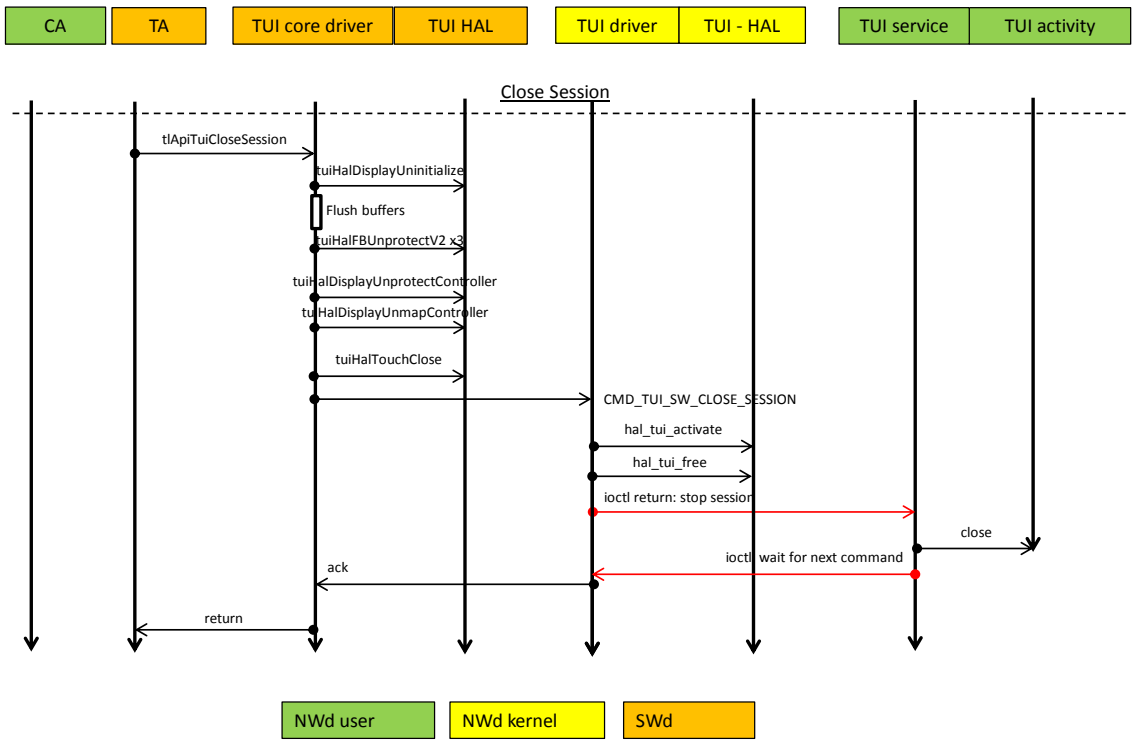


Figure 10: TUI flow chart: session closing.

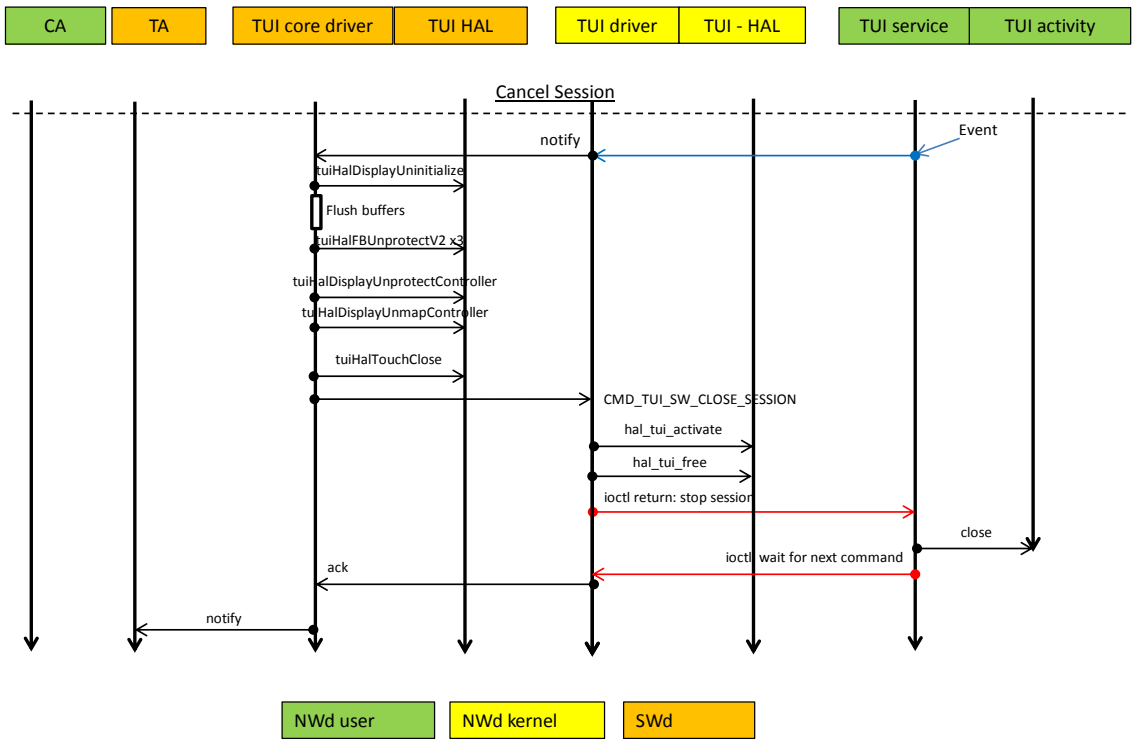


Figure 11: TUI flow chart: session cancellation.

7 VALIDATING THE PRODUCT

7.1 BASIC VERIFICATION AND VERSION CHECK

The Kinibi product comes with several samples and informative applications. It is recommended to install and run them to verify basic functionality and to check the Kinibi version.

To verify that the basic functionality works, follow this integration guide and then boot the device. This should have installed the Content Management Trusted Application *(tlcm.axf)* `07010000…0000.tlbin` in `/system/app/mcRegistry`. This TA will be used to test that TA loading works and to get the Kinibi version. To do so, you need to install the `tlcInfo` tool that is provided in the product package.

Installing the `tlcInfo` tool:

```
$ adb push t-base-dev-kit/Tools/Info/<TOOLCHAIN>/<MODE>/tlcInfo
/data/misc
```

Running the `tlcInfo` tool:

```
$ cd /data/misc/
$ ./tlcInfo
```

Expected output:

```
Copyright (c) Trustonic Limited 2013-2015.
Test-Package-ARNDALE-311A-20160831_184510_13773_45507, Aug 31 2016,
19:29:18.

mcGetMobiCoreVersion:
productId        = t-base-EXYNOS64-Android-311A-V004-20160527_225213_11082_
38854
versionMci       = 0x00010005
versionSo        = 0x00020002
versionMclf      = 0x00020005
versionContainer = 0x00020001
versionMcConfig  = 0x00000003
versionTlApi     = 0x00010013
versionDrApi     = 0x00010004
versionCmp       = 0x00060001

CMP GetSuid:
SUID             = 0x020000001000052438 42856AC76010000
SUID.SiP         = 0x00000002
SUID.SoC         = 0x43520010
SUID.CSN         = 0x842856AC76010000
CMP GetVersion:
productId        = t-base-EXYNOS64-Android-311A-V004-20160527_225213_11082_
38854
versionMci       = 0x00010005
versionSo        = 0x00020002
versionMclf      = 0x00020005
versionContainer = 0x00020001
versionMcConfig  = 0x00000003
```

```
versionTlApi      = 0x00010013
versionDrApi      = 0x00010004
versionCmp        = 0x00030000
```

Note the Kinibi product version in productId field that must match the version from the release notes and the release zip file name.

7.2 RUNNING THE TTS

The Kinibi product comes with the TTS apk, the Trustonic Test Suite which contains tests to validate the Kinibi integration on the platform.

The test suite is located in the `TTS` folder that contains complete documentation explaining how to run the tests.

8 System Debugging

During integration of the various components, there have been a couple of issues that appear rarely and cannot be reproduced when being observed (sometimes called *Panda bugs*). The only way to find and fix such bugs is to have enough information about the system history and the current state of affairs at the moment of the crash.

8.1 LINUX DEBUG FS

Linux supports a *debugfs* filesystem, usually mounted under `/sys/kernel/debug/`, but also under `/d/` on Android. See http://en.wikipedia.org/wiki/Debugfs for more details.

In Kinibi-310A, the NWd driver will create a `trustonic_tee` directory in the debugfs tree that holds the following files.

Debugfs file	access rights	Contents

`crashdump`	`read-only, root-only`	A copy of the crash dump that also appears in the kernel log on TEE crash or hang. Only created when a crash or hang occurs.
		
		Content example:
		`flags = 0x80000401`
		`haltCode = 0x00000002`
		`haltIp = 0x07f020d6`
		`faultRec.cnt = 0x00000003`
		`faultRec.cause = 0x00000002`
		`faultRec.meta = 0x00000000`
		`faultRec.thread = 0x00010005`
		`faultRec.ip = 0x000010e6`
		`faultRec.sp = 0x00007da0`
		`faultRec.arch.dfsr = 0x00000807`
		`faultRec.arch.adfsr = 0x00000000`
		`faultRec.arch.dfar = 0x00000000`
		`faultRec.arch.ifsr = 0x00000005`
		`faultRec.arch.aifsr = 0x00000000`
		`faultRec.arch.ifar = 0xbe001070`
		`mcData.flags = 0x00000001`
		`mcExcep.partner = 0xffffffff`
		`mcExcep.peer = 0x00010005`
		`mcExcep.cause = 0x00000002`
		`mcExcep.meta = 0x00000000`
		`mcExcep.uuid = 0x02040000000000000000000000`

`last_mcp_commands`	`read-only, root-only`	List of last MCP commands (limited to 256).
		`PID command S-ID state result errno`
		`1928 get version 0 complete 0 0`
		`1928 open session 301 complete 0 0`
		`1928 load token 0 complete 0 0`
		`1932 open session 401 complete 0 0`
		`2983 open session 501 complete 0 0`
		`2983 close session 501 complete 0 0`
		`2998 open session 501 complete 0 0`
		`2998 close session 501 complete 0 0`
		`3033 open session 501 complete 0 0`
		`3033 close session 501 complete 0 0`

<table>
<tr><td></td><td></td><td>

```
 3033 open session   501 complete      0     0
 3033 close session  501 failed        0    62
```

</td></tr>
<tr><td>last_smc_commands</td><td>read-only, root-only</td><td>List of last SMC commands (limited to 256). Only created when a crash or hang occurs.

```
CPU clock    command param1     param2     param3
8918131208 -1 0x5de7e000 0x00000118 0x011800a0
8918142083  4 0x00000000 0x00000000 0x00000000
8918317917 -2 0x00000000 0x00000000 0x00000000
8918326292  3 0x00000000 0x00000000 0x00000000
8918690458  3 0x00000000 0x00000000 0x00000000
```

</td></tr>
<tr><td>sessions</td><td>read-only, root-only</td><td>List of sessions known to the NWd.

```
  ID type   ec state
 301 MC      0 running
 401 GP      0 running
```

ec: error code

Session type MC: legacy session

Session type GP: GP session</td></tr>
<tr><td rowspan="2">structs</td><td>read-only, root-only</td><td>Hierarchical view of the clients, sessions, buffers, mappings and MMU tables.</td></tr>
<tr><td colspan="2">Content example:

```
client de084680 [2]: [kernel]
  cbuf dd862f80 [1]: addr dd85d000 uaddr 0 len 4096
  session de90bb00 [1]: 301 ec 0
    wsm dd862fc0: mmu de341000 cbuf dd862f80 va dd85d000 len 4096
      mmu de341000: kernel len 4096 off 0 table dd866000 type L2
client dde57000 [1]: McDaemon.FSD (1935)
  session de957e00 [2]: 401 ec 0
    wsm ddfceac0: mmu de899000 cbuf   (null) va 403ad008 len 1024
      mmu de899000: user len 1024000 off 8 table dd864000 type L2
client dd7b3f80 [2]: INTEGRATION_Bla (2624)
  cbuf ddf97a40 [2]: addr dcd90000 uaddr 403e9000 len 4096
  session dd780200 [1]: 501 ec 0
    wsm dd6d9d40: mmu dccb6000 cbuf ddf97a40 va 403e9000 len 408
      mmu dccb6000: kernel len 40 off 0 table dcd8d000 type L2
```

Description:

```
client <struct pointer> [<number of users>]: <thread name>
(<PID>)
   cbuf <struct pointer> [<number of users>]: addr <kernel
virtual address> uaddr <user-space virtual address> len
<length>
   session <struct pointer> [<number of users>]: <ID> ec
<exit code>
     wsm <struct pointer>: mmu <pointer to mmu> cbuf <pointer
to cbuf> va <virtual address> len <length>
       mmu <struct pointer>: <buffer type> len <length> off
<offset> table <pointer to MMU table> type L<MMU type>
```

</td></tr>
<tr><td>structs_counter</td><td>read-only, root-only</td><td>Counters of structures created and not freed, to check for memory leaks.

```
clients:  2
cbufs:    1
sessions: 2
wsms:     2
mmus:     3
```

</td></tr>
</table>

swd_debug	read/write, root-only	A boolean entry to enable or disable the SWd logging into the kernel log.

In Kinibi-310B, the following additional entries are available:

active_cpu	write-only, root-only	An integer entry to trigger a core switch of the SWd to the specified cpu.
mcp_timeout	read/write, root-only	An integer to set the MCP timeout in seconds. The default value is 10. This timeout is repeated 5 times until the MCP is declared dead. In debugging sessions this feels like a watchdog bite after 50 seconds. Example for 10 hours timeout: `echo 36000 > /d/trustonic_tee/mcp_timeout`

8.2 TEE DEBUGSESSION

Kinibi-311A adds more system debugging support by the addition of tee-ps tool and TEE DebugSession infrastructure. The product package contains **tee-ps** tool in **/t-base-dev-kit/Tools/TlcTeePs** with source code and binaries.

Running tee-ps requires root access and allows to the system integrator to see snapshots of the state of the TEE. In particular, the tool shows the list of sessions including session ID, UUID, memory usage and associated threads. For each thread, tee-ps shows the threadid, last syscall, user time and system time.

Developers can run tee-ps in various modes and display different kinds of information. The design of the TEE DebugSession infrastructure ensures that tee-ps can access the information even if the TEE is crashed or in dead-lock. During Kinibi integration phase, the TEE DebugSession could be tweaked to trace additional debug information.

Sample Output:

```
root@android:/data/misc # ./tee-ps -g
Local time: 2000-02-08 @ 05:29:03 mcDriverDaemon PID: 2574 Size of debug info: 43008

TEE Product Id: Build by syltro01@str-hp840g2 01/22/16 15:56:42_29352

TEE kernel info:
 Max threads=200, current thread=0xffffffff, last thread=0x00020001, last
syscall=IPC(17)

000: System TA
 SPID=0xfffffffb, UUID=12341234-1234-1234-1234-123412341234
 State=Active, flags=0x00000001, max instances=0
 Uses a total of 157 pages (628 KB), 4 tables:
  BlobRO=0/0, BlobRW=41/164, Client=0/0,
  WSM=0/0, Phys=0/0, Heap=0/0, Misc=104/416
 IPCH info: ID=8, len pages=0, mr0=0x701, mr2=0xfffffffb

        101 Thread Id=0x00010001 at PC=0x00002490, SP=0x000283c0
        flags=0x00dd000e, ipcPartner=0x00000000, last syscall=IPC(17),
        last modified=119956955 ms, total user time=12131 µs, total sys time=48161 µs

        102 Thread Id=0x00020001 at PC=0x00010414, SP=0x00028730
        flags=0x005d000c, ipcPartner=0xffffffff, last syscall=IPC(17),
        last modified=120045463 ms, total user time=495 µs, total sys time=698297 µs

        103 Thread Id=0x00030001 at PC=0x0000563e, SP=0x00028e3c
        flags=0x00dd000d, ipcPartner=0x00000000, last syscall=IPC(17),
        last modified=120018985 ms, total user time=38510 µs, total sys time=676368 µs
```

```
        104 Thread Id=0x00040001 at PC=0x0000f51c, SP=0x00029660
        flags=0x0000010c, ipcPartner=0x00000000, last syscall=SIGNAL_WAIT(20),
        last modified=118587839 ms, total user time=2248 µs, total sys time=52708 µs

201: Driver, ID=0x150/336
 SPID=0x00000000, UUID=02010000-0000-0000-0000-000000005000
 State=Active, flags=0x0000000b, max instances=1
 Uses a total of 39 pages (156 KB), 1 tables:
  BlobRO=0/0, BlobRW=32/128, Client=0/0,
  WSM=0/0, Phys=0/0, Heap=21/84, Misc=0/0
 IPCH info: ID=2, len pages=0, mr0=0x10003, mr2=0x0

        202 Thread Id=0x00020002 at PC=0x07d0b8ea, SP=0x00039fd0
        flags=0x005d0004, ipcPartner=0x00030001, last syscall=IPC(17),
        last modified=119800226 ms, total user time=3438 µs, total sys time=81860 µs

301: Driver, ID=0x104/260
 SPID=0x00000000, UUID=07050500-0000-0000-0000-000000000020
 State=Active, flags=0x0000000a, max instances=1
 Uses a total of 34 pages (136 KB), 2 tables:
  BlobRO=0/0, BlobRW=15/60, Client=10/40,
  WSM=0/0, Phys=0/0, Heap=9/36, Misc=0/0
 IPCH info: ID=2, len pages=0, mr0=0x10007, mr2=0x0

        301 Thread Id=0x00010003 at PC=0x07d0b8ea, SP=0x0001bfc8
        flags=0x005d0009, ipcPartner=0x00030001, last syscall=IPC(17),
        last modified=119892360 ms, total user time=3373 µs, total sys time=96692 µs

401: System TA
 SPID=0xfffffffe, UUID=07050500-0000-0000-0000-000000000022
 State=Dead, flags=0x00000008, max instances=1
 Uses a total of 4 pages (16 KB), 1 tables:
  BlobRO=0/0, BlobRW=2/8, Client=0/0,
  WSM=0/0, Phys=0/0, Heap=2/8, Misc=0/0
 IPCH info: ID=0, len pages=0, mr0=0x0, mr2=0x0

501: System TA
 SPID=0xfffffffe, UUID=07050501-0000-0000-0000-000000000020
 State=Active, flags=0x00000008, max instances=1
 Uses a total of 14 pages (56 KB), 2 tables:
  BlobRO=0/0, BlobRW=3/12, Client=0/0,
  WSM=256/1024, Phys=0/0, Heap=1/4, Misc=0/0
 IPCH info: ID=27, len pages=10, mr0=0x104, mr2=0x1004fc

        501 Thread Id=0x00010005 at PC=0x07d02552, SP=0x000040c0
        flags=0x00000104, ipcPartner=0x00000000, last syscall=SIGNAL_WAIT(20),
        last modified=119846168 ms, total user time=12 µs, total sys time=136 µs

601: System TA
 SPID=0xfffffffb, UUID=08050500-0000-0000-0000-000000000000
 State=Active, flags=0x00000000, max instances=1
 Uses a total of 157 pages (628 KB), 4 tables:
  BlobRO=0/0, BlobRW=41/164, Client=0/0,
  WSM=256/1024, Phys=0/0, Heap=0/0, Misc=104/416
 IPCH info: ID=0, len pages=0, mr0=0x0, mr2=0x0

701: System TA
 SPID=0xfffffffe, UUID=07050500-0000-0000-0000-000000000030
 State=Dead, flags=0x00000008, max instances=1
 Uses a total of 257 pages (1028 KB), 2 tables:
  BlobRO=0/0, BlobRW=0/0, Client=0/0,
  WSM=0/0, Phys=0/0, Heap=0/0, Misc=0/0
 IPCH info: ID=0, len pages=0, mr0=0x0, mr2=0x0
```

9 RPMB INTEGRATION

Kinibi protects against the following rollback attacks:

- Replacing a Secure Filesystem partition with an old version
- Replacing a System TA with an old version

See Figure 12 for the different components and partitions:

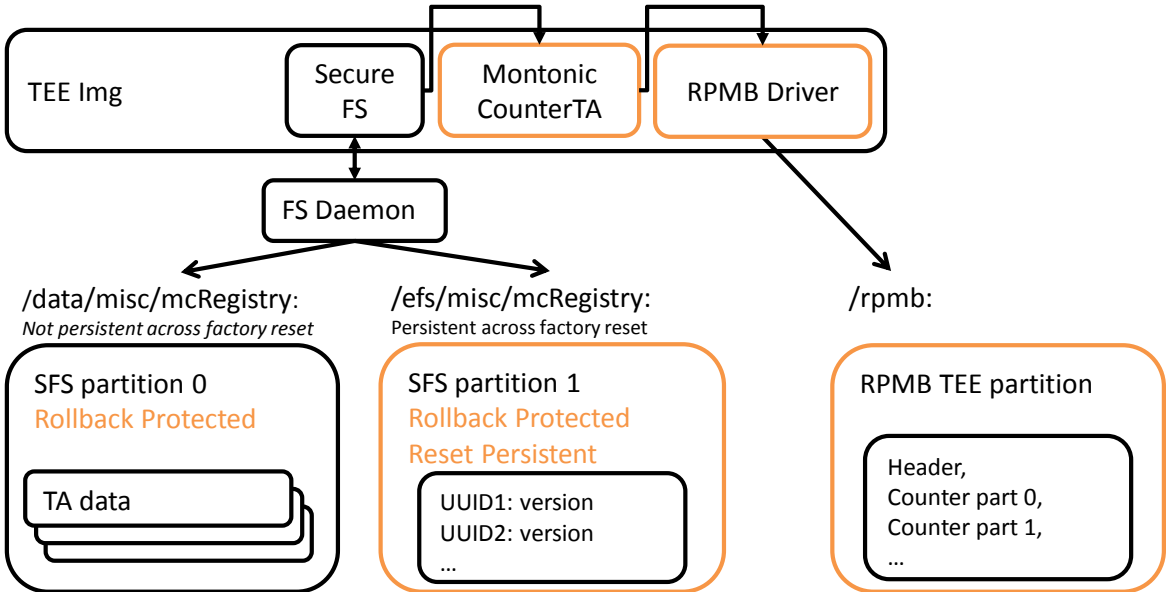


Figure 12: Secure Storage System

9.1 SECURE STORAGE SYSTEM

The Kinibi Secure Filesystem provides the GlobalPlatform Trusted Storage API to Trusted Applications and Secure Drivers. The Secure Filesystem handles file accesses using a sophisticated B+ tree that allows an efficient retrieval of the persistent objects with a reduced number of I/O operations. The underlying partitions are stored in the Normal world with the help of the Filesystem Daemon (FSD). The FSD is part of the Kinibi Daemon and connects to the Secure Filesystem at Daemon startup. FSD will then wait for requests from the TEE to modify the partition backend files.

Kinibi-400A supports two partitions:

- partition 0: user data and TA binaries
  - persistent objects accessed via the GlobalPlatform Trusted Storage API
  - TA binaries installed via TAMv2
  - user-specific
  - wiped during factory reset
  - e.g. “/data/misc/mcRegistry”
  - Daemon parameter -p <path>
  - Can be TEE rollback protected
- partition 1: System TA version information
  - version information for system TAs that are not using the GlobalPlatform API
  - system-specific
  - persist factory reset
  - e.g. “/efs/misc/mcRegistry”
  - Daemon parameter –P1 <path>

- Must be TEE rollback protected
- Must be TEE factory-reset persistent

9.2 CONFIGURING KINIBI IMAGE FOR RPMB USE CASES

By default, the RPMB support in Kinibi image is disabled.

The integrator has to configure the Kinibi image with the desired RPMB behavior using MobiConfig. The MobiConfig Rollback Protected Storage(--rps) mode is based on masks and can set 2 flags for up to 16 partitions.

Currently there are two masks:

MobiConfig option	Behavior
`--use_mask`	Enable Rollback Protection for partitions in mask. When the Rollback Protection is activated, the Secure Filesystem recognizes if the backend partition was rolled back to a previous state and returns respective error codes in the GlobalPlatform Trusted Storage API. Note that removal of the partition leads to recreation of a new empty partition.
`--reset_persistent_mask`	Enable Reset Persistence for partitions in mask. This flag, which only works in combination with the use_mask, marks a partition as a system partition that must persist across factory resets. When Secure Filesystem creates this partition for the first time, it links the partition to an RPMB counter forever. If the NWd removes, corrupts or uses an older version of the partition backend file, the Secure Filesystem will return error codes to any access to files in this partition. The only way to recover is to revert to the correct backend file.

9.2.1 Configure TA downgrade protection for partition 1 and RP for partition 0:

The integrator has to configure the TEE image with MobiConfig:

```
$ cd SecureIntegration/tools/MobiConfig
$ java -jar Bin/MobiConfig.jar
    --rps
    -i tee.img
    -o tee_rps.img
    --use_mask 3
    --reset_persistent_mask 2
    --chunk-id 0
```

9.2.2 Configure rollback protection for partition 0:

The integrator has to configure the TEE image with MobiConfig:

```
$ cd SecureIntegration/tools/MobiConfig
$ java -jar Bin/MobiConfig.jar
    --rps
    -i tee.img
    -o tee_rps.img
    --use_mask 1
    --chunk-id 0
```

For more information about MobiConfig, see Appendix II.

9.3 INTEGRATING RPMB HARDWARE

For the Kinibi Rollback Protection to work, the device must have an RPMB-compatible peripheral and Kinibi must access it. The integrator has to implement a custom secure channel between the RPMB driver and the RPMB peripheral. The RPMB driver must be adapted to the Kinibi Secure Filesystem.

Kinibi has two ways to integrate the platform RPMB driver:

- Use the skeleton DrRPMB driver and implement the Trustonic RPMB interface
- Use its own RPMB driver and adapt the Monotonic Counter TA

The Monotonic Counter Proxy TA is used to communicate between the DrRPMB and the Secure Filesystem Driver. That way the SFS driver can read and write counter values.

9.3.1 Integrating RPMB Driver

The product package contains the **DrRPMB** sample driver in SecureIntegration/rpmb/DrRPMB directory.

DrRPMB Secure Driver UUID

The DrRPMB UUID is :

0705-0500-0000-0000-0000-0000-0000-0021

DrRPMB driver ID

The DrRPMB Driver ID is : 0x0700

Commands

The DrRPMB must implement the following commands:

```
RPMB_MESSAGE_TYPE_OPEN        1
RPMB_MESSAGE_TYPE_CLOSE       2
RPMB_MESSAGE_TYPE_WRITE       3
RPMB_MESSAGE_TYPE_READ        4
```

Marshalling Parameters Structure

The driver must implement the following interfaces:

```
typedef struct {
    uint32_t size;    /* Size of the exchange buffer message: 16 + n */
    uint16_t version;  /* Must be 0. Reserved for future extensions */
    uint16_t command;     /* Command identifier */
    uint32_t status;      /* Return status */
    uint32_t handle;      /* Object handle */
    union {
        struct {
            addr_t   data;
            uint32_t data_size;
        }
        cmdRPMBOpen_t cmdRPMBOpen;
        rspRPMBOpen_t rspRPMBOpen;
        cmdRPMBRead_t cmdRPMBRead;
        rspRPMBRead_t rspRPMBRead;
        cmdRPMBWrite_t cmdRPMBWrite;
        rspRPMBWrite_t rspRPMBWrite;
    } payload;
} rpmbExchangeBuffer_t,*rpmbExchangeBuffer_ptr;
```

To start a new integration, Trustonic will provide an additional document explaining the interface in more detail.

Startup integration

The integrator should start the RPMB driver before starting any other TA or Driver that requires the Rollback protection.

9.3.2 Integrating Monotonic Counter TA

The product package contains the TA_Counter skeleton code in SecureIntegration/rpmb/ta_counter directory.

TA Counter UUID

The TA_Counter UUID is :

0705-0500-0000-0000-0000-0000-0000-0022

TA Counter Porting

There are two sample implementations:

File name	Behavior
`Locals/Code/rpmb_counter1.c`	Uses the Trustonic RPMB interface of DrRPMB.
`Locals/Code/rpmb_counter2.c`	Uses the RPMB interface based on `rpmb_interface_skeleton.h`
`Locals/Code/rpmb_interface_skeleton.c`	Dummy implementation of tlApiRpmbRead() and tlApiRpmbWrite()

The integrator chooses the implementation that fits best its DrRPMB, adapts the Makefile as required and implements the respective interface.

TEE Image integration

The TA Counter is part of the TEE image and started automatically by the TEE at boot. The integrator has to use the TEE image builder to include the platform-specific TA Counter in the TEE image (See 9.3.3). The image builder will take the following file to include in TEE image:

```
SecureIntegration/t-base-kit/Locals/Objects/ta-
mctl/ARM_V7A_STD/$(MODE_DIR)/07050500000000000000000000000022.tlbin
```

Startup integration

The TEE image starts the TA Counter automatically.

The integrator should start the RPMB driver before starting any other TA or Driver that requires the Rollback protection.

9.3.3 TEE Image builder

The product package contains the TEE image builder in in SecureIntegration/t-base-kit/ directory.

The content of the directory should be as follows:

```
t-base-kit/
    Locals/
      Build/
      Code/
      Objects/
        driver-crypto/
        driver-sth2/
        driver-sthu/
        CHIP_ID/
        ta-mctl/
        ta-sth2proxy/
    Out/
```

The respective folders in Objects/ contain the stripped ELF files of the components that make up the Kinibi TEE image.

To build the image, you need to call the build script:

```
$ cd SecureIntegration/t-base-kit
$ ./Locals/Build/build.sh
```

This should create the TEE image in:

```
SecureIntegration/t-base-kit/Locals/Objects/$PLATFORM/$MODE/tee.img
SecureIntegration/t-base-kit/Locals/Objects/$PLATFORM/$MODE/t-base.img
```

The created objects should be the same as:

```
SecureIntegration/t-base/Bin/MobiCore/$PLATFORM/$MODE/tee.img
```

The resulting image then has to undergo configuration as described in other parts of this guide.

The TEE image builder is based on python and the python ELF library from:

https://github.com/eliben/pyelftools

On an Ubuntu/Debian system, you can install them with the following command:

```
$ sudo apt-get install python-pip
$ sudo pip install pyelftools
```

9.4 SYSTEM TA DOWNGRADE PROTECTION

Kinibi protects System TAs against downgrade attacks. Respective TAs and Secure Drivers must be specifically marked with a version and the downgrade-protection flag. Kinibi will then automatically enroll them, automatically upgrade the last known version and fail the mcOpenSession command if an attacker downgrades the TA.

The integrator has to do the following steps to enable TA downgrade protection:

1. Integrate an RPMB driver (see 9.3)
2. Configure Partition 1 to be reset persistent and rollback protected (see 9.2.1)
3. Chose a location for Partition 1 that persists factory reset and tell the Daemon -P1, see 2.2.3.3.2
4. Inject the version and dp flag into the TAs (see 9.4.1)

9.4.1 Marking TAs for downgrade protection

A new option has been added to MobiConvert and the TlSdk and DrSdk makefiles.

New MobiConvert option:

```
-dp, --downgrade-protection: activate downgrade protection for the TA
```

New TlSdk option:

```
TA_ROLLBACK_PROTECTED:=Y
```

New DrSdk option:

```
DR_ROLLBACK_PROTECTED:=Y
```

Sample call to create a downgrade protected system TA with version 0.1:

```
java -jar MobiConvert.jar -b <pathToBinary> -o <pathToOutput> -k
<pathToKeyFile> -s 3 --downgrade-protected --interface-version 0.1
```

Appendix I. Google Compliance Test Suite

1 Details of the CTS waiver request

In general any Android application that wants to communicate with a Trusted Application running in Kinibi needs to be able to share read/write memory with the Trusted Application. Applications can be downloaded from the Google Play Market with the functionality to communicate with a Trusted Application running in Kinibi.

The only way to share memory between an Android Application and a Kinibi Trusted Application is through the Linux device node `/dev/mobicore-user`. The Android application needs to send at least 2 calls to the Linux kernel to prepare shared memory buffers (L2 MMU tables) for Kinibi, an IOCTL call and a mmap (`PROT_READ | PROT_WRITE`) call. Because of these 2 calls the Android application needs Read & Write access to the `/dev/mobicore-user` device node.

Because we assume any application from the Google Play Market should be allowed to communicate with the Secure World there is no way to assign a Group ID (GID) to the application in order for it to access Kinibi. Because the node cannot assume any GID for all the applications the node needs the extended permissions of `0666`.

Please note that write is only required for the mmap (PROT_READ | PROT_WRITE) call as this is a definition in the Linux kernel. The write method is not implemented for the `/dev/mobicore- user` and no write is actually allowed from user space.

This method of accessing the Secure World through the /dev/mobicore-user device node has been fully approved by Google and a waiver is in place for their Compliance Test Suite (CTS). Details of Google's approval process for this waiver can be found at:

https://android-review.googlesource.com/#/c/63940/

The code of the CTS where this waiver is implemented can be found at:

https://android.googlesource.com/platform/cts/+/3e29e6b0916db3f4b59657dabde9a40815244097/tests/tests/permission/src/android/permission/cts/FileSystemPermissionTest.java

Appendix II. MobiConfig Manual

Help output

```
java -jar MobiConfig.jar
MobiConfig V3.3 Build by lukhan01@DE0015 01/26/16 17:55:13
  Copyright (c) 2013-2015 TRUSTONIC LIMITED. All rights reserved.

java -jar MobiConfig.jar <command> <options>...

<command>
  -h,--help                    Show help
  -c,--config <config-opts>  Configuration options, see below
  -p,--preconfig <preconfig-opts> Pre-configuration options, see below
  --rps Rollback-protected storage configuration options, see below
  -d,--dump <image>      Dump configuration settings of image and exit
  -de,--dump-endorsement   <image>   Dump   endorsement   configuration
settings of image and exit
  -ds,--dump-sipid <image> Dump Silicon provider ID of image and exit
  -dr,--dump-rps <image> Dump Rollback-protected storage configuration

<config-opts>
  -i,--in <in-image> -o,--out <out-image>
  [-k,--key <keyfile> [--kid <keyid>]  |
   [--kphkey1 <keyfile1> --rekey1 <rekeyfile1> --kid1 <keyid1>
    --kphkey2 <keyfile2> --rekey2 <rekeyfile2> --kid2 <keyid2>
  ]]
  -ek,--ekey <ekeyfile>  | [-xc,--xml-config <xml-manifest> ]

<preconfig-opts>
  -i,--in <in-image> -o,--out <out-image> -ek,--ekey <ekeyfile> -si,--
sipid <sipid>-df,--dflags <dflags>
  <keyfile>      PEM file containing public key
  <keyid>        Puk.Kph.Request Key-ID (keyid >= 1; default:1)
  <ekeyfile>     PEM file containing endorsement server public key
  <sipid>        Silicon provider ID
  <xml-manifest>  XML  file  containing  the  key  list  with  access
permission flag and reserved uuid list

<Rollback-protected storage configuration options>
  --use_mask <Rollback-protected storage for a partiton enabled mask>
  --reset-persistent_mask <Blanking by erasure configuration mask>
  --chunk-id <First Rollback-protected storage chunk ID>
   <in-image>     Input image file
   <out-image>    Configured output image file

Device binding key-related options:
  <kphkeyfile[n]>   PEM file containing Device Binding request signing
key
                    (PuK.Kph.Request) public key at table entry n.
  <rekeyfile[n]>          PEM   file   containing   Receipt   Encryption
(PkP.Root.Transport)
```

```
                    public key at table entry n.
  <keyid[n]>     Key ID for table entry n;
                 defaults to Key ID 1 (Trustonic).

                 Note that use of multiple table entries requires
                 a version of CMTL that supports the structure.
                 If n is not specified, it defaults to table entry #1.
```

I. Multi OEM keys

Since Kinibi-311A, the SIP and the OEM can configure up to 32 OEM keys into the Kinibi image.

The MobiConfig tool has a new option --xml-config that accepts an XML file with a list of keys and options:

```
<?xml version="1.0"?>
<parameters>
<!-- manifest used for test -->
  <oem-keys>

    <oem-reserved-uuid-list>
        <!-- list with up to 16 different UUIDs:
            Allow the OEM to restrict a certain key to a limited number of UUIDs.
            To make use of this list, define a key without the ALLOW_ANY_UUID flag.
         -->
        <uuid value="08010000000000000000000000000011"/>
    </oem-reserved-uuid-list>

    <key-list>
        <!-- list with up to 32 different hashes:

             <key filename=... flags=.../>
             <key hash=... flags=.../>

Attributes:
"filename": is a path to a PEM file, where the hash will be generated over the public
key
"hash":     a hexadecimal byte string of the form "xxxx...xx", length is 64 chars,
each char must be [0-9,A-F,a-f]
"flags":    case insensitive comma separated list of
        "ALLOW_DRIVER",   # This key signs drivers
        "ALLOW_MIDDLEWARE # This key signs middle ware
        "ALLOW_SYSTEM_TA" # This key signs system TA
        "ALLOW_ANY_UUID"  # This key signs any UUID, otherwise UUID must NOT be in
<oem-reserved-uuid-list>

Either "hash" or "filename" should be used. If both attributes present, "hash" must
match the hash generated from the PEM file. If not, the program will abort with an
error. This can be used as a fail-safe checkto ensure the correct PEM file is used and
there is a key pair for a hash.
         -->
        <key filename="key_driver_pub.pem" flags="ALLOW_DRIVER"/>
        <key filename="pairVendorTltSig.pem"
flags="ALLOW_DRIVER,ALLOW_MIDDLEWARE,ALLOW_ANY_UUID,ALLOW_SYSTEM_TA"/>

    </key-list>

  </oem-keys>
</parameters>
```