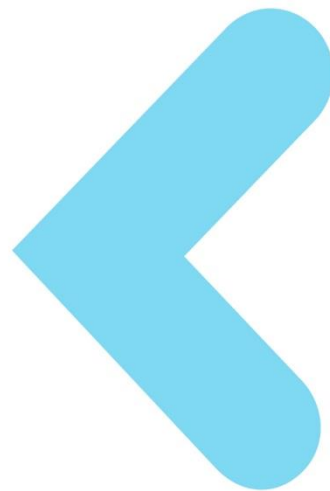
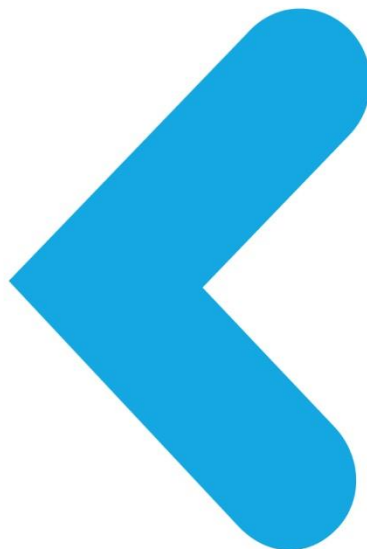


INTEGRATION GUIDE

◀t-base
Integration Guide



PREFACE

This specification is the confidential and proprietary information of Trustonic ("Confidential Information"). This specification is protected by copyright and the information described therein may be protected by one or more EC patents, foreign patents, or pending applications. No part of the Specification may be reproduced or divulged in any form by any means without the prior written authorization of Trustonic. Any use of the Specification and the information described is forbidden (including, but not limited to, implementation, whether partial or total, modification, and any form of testing or derivative work) unless written authorization or appropriate license rights are previously granted by Trustonic.

TRUSTONIC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF SOFTWARE DEVELOPED FROM THIS SPECIFICATION, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. TRUSTONIC SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SPECIFICATION OR ITS DERIVATIVES.

VERSION HISTORY

Version	Date	Modification
1.0	January 5 th , 2013	First version
1.1	May 22d, 2013	Update for <t-base-202
1.2	November 21 st , 2013	Updated for <t-base-300
1.3	January 14 th , 2014	Added missing information for drivers signing.
1.4	March 11 th , 2014	Added corrections and clarifications for fastcall hooks, <t-play and Trusted UI integration.
1.5	May 13 th , 2014	Fix font inconsistencies
1.6	June 9 th , 2014	Updated for <t-base-301
1.7	July 9 th , 2014	Added missing information for the location of the keymaster in the Android File System.
1.8	July 22 nd , 2014	Fixed the error codes returned by the DRM driver.
1.9	July 24 th , 2014	Security warnings added in TUI integration sections
2.0	September 20 th , 2014	Integration information for Android 64 bit OS added

TABLE OF CONTENTS

1	Introduction	6
2	<t-base Normal World Components.....	7
2.1	Building Normal World Components	8
2.1.1	<t-base User-space Components	8
2.1.2	<t-base Kernel-space Components	8
2.2	Integration in Android (32 and 64 bit)	10
2.2.1	Integration in Device File System.....	10
2.2.1	Permissions and Access Rights	11
2.2.2	Directories Requirements.....	11
2.2.3	Starting up <t-base	12
3	<t-base Secure World Components	13
3.1	<t-base in Boot Chain	14
3.2	Trusted Applications and Secure Drivers Overview	15
3.3	Config and Signing of t-base Components.....	16
3.3.1	Keys in <t-base Environnent.....	16
3.3.2	Configuring <t-base Binary	17
3.3.3	Configuring Content Management Trusted Application	18
3.3.4	Signing the Secure Storage Driver	19
3.3.5	Signing the Trusted User Interface Driver	19
3.3.6	Signing the DRM Driver	20
3.3.1	Signing the TEE Keymaster Trusted Application	20
3.4	<t-base Boot Parameters on 32 bit Hardware.....	21
3.5	<t-base Boot Parameters on 64 bit Hardware.....	22
3.6	Fastcalls Hook Mechanism.....	23
3.6.1	Handling FastCalls	24
3.6.2	Additional Secure Driver APIs	25
3.6.3	Firmware Driver Structure	29
3.6.4	u-boot Integration Sample	31
4	TEE Keymaster	36
4.1	Normal World Connector and Integration	37
4.2	Secure World Component	38
5	DRM Integration	39
5.1	High Level Flow	39
5.2	T-Play Assumptions.....	40

5.3	Drivers Overview	40
5.3.1	Framework Support	40
5.3.2	TLC and TA Driver Access	41
5.3.3	Driver-Client Access Control	41
5.3.4	Threads	41
5.3.5	Protected Buffers	42
5.4	DRM Driver Protocol	42
5.5	Security and Evaluation Considerations	45
5.5.1	Video Buffer Protection	45
5.5.2	Checking of Pointers	46
5.5.3	Input to Crypto Hardware	46
5.5.4	Integrity of System Components	46
5.5.5	Trusted Application Isolation	46
5.5.6	Debug Attack	46
5.5.7	Reset Buffers	46
6	Trusted User Interface Integration	47
6.1	Security considerations.....	48
6.1.1	Framebuffer	48
6.1.2	Input devices	48
6.2	TUI Secure Driver	49
6.2.1	Memory requirement.....	49
6.2.2	Secure display	50
6.2.3	Secure input	50
6.2.4	Building the TUI secure driver	51
6.2.5	Integrating the TUI secure driver.....	51
6.3	TUI Kernel components	52
6.3.1	TUI module	52
6.3.2	Patching Linux drivers	53
6.3.3	Building the kernel.....	53
6.3.4	Integrating the TUI module.....	55
6.4	TUI Android components.....	55
6.4.1	Customizing the TUI Service	55
6.4.2	Integrating the TUI service	55
7	Running Tests	56

1 INTRODUCTION

This document explains how to integrate <t-base on a platform. <t-base integration consists in:

- ◀ Integrating the <t-base Normal World components for Android.
- ◀ Integrating the <t-base Secure World components.
- ◀ Integrating the Android Keymaster modules for <t-base.
- ◀ Implementing support of <t-play.
- ◀ Implementing <t-base Trusted User Interface (TUI).
- ◀ Running the Validation Test Suite to verify the integration.

Note that in order to enable <t-base-3xx and the OTA-Operated Containers on a Device, the device manufacturer must install and integrate Trustonic's Key Provisioning Host (KPH) at its manufacturing line. The KPH is a tool which injects the TEE Authentication Token on the device and which stores a copy in the Trustonic backend system. This process and the integration steps are specified in a separate document (*t-base_Provisioning_Manual.pdf*).

The <t-base product package is structured as follows:

- ◀ *Documentations*, this directory regroups the integrators documentations (how to integrate <t-base into a device, how to use <t-base provisioning tools).
- ◀ *AndroidIntegration*, the directory which contains all the Normal World components.
- ◀ *SecureIntegration*, the directory which contains all the Secure World components (<t-base core binary and Trustonic System Trusted Applications).
- ◀ *t-base-dev-kit*, the <t-base development kit, with documentations for Trusted Applications developers and samples.
- ◀ *ValidationTestSuite*, a basic test suite which will allow integrators to quickly validate that all the critical features of <t-base have been successfully integrated.

2 <t-base NORMAL WORLD COMPONENTS

Normal World components of <t-base are:

In Android User space:

- ◀ The <t-base Daemon: component in charge of registry and initialization of Normal World/Secure World communication.
- ◀ The Root Provisioning Agent: component that is used when installing service provider and developer trusted applications (it includes the Curl library, for XML manipulations and a log library).
- ◀ The <t-base Client library: user space APIs available for Normal World application developers.
- ◀ The <t-base registry interface: management of the registry.
- ◀ The <t-base Keymaster library: interface to access to Android Keymaster feature.

In Linux Kernel space:

- ◀ The <t-base Linux driver: component in charge of communication between the Normal World user space client and the Secure World (with two interfaces available: one reserved for the Daemon and one allocated to direct communication with all the user space clients).
- ◀ The <t-base Kernel API: kernel level APIs available for Normal World driver developers.

All the Normal World components of <t-base product are provided in source code and in binary, with support of 32 or/and 64 bit versions. They are all grouped in directory *AndroidIntegration*.

The purpose of the *AndroidIntegration* folder is to provide a simple way of integrating the Normal World components of <t-base in an Android source tree.

2.1 BUILDING NORMAL WORLD COMPONENTS

The *AndroidIntegration* folder contains 2 parts:

- ◀ A *mobicore* subdirectory that groups all the Android user space components (binaries and libraries).
- ◀ And a *gud* subdirectory which contains all the Android kernel components.

2.1.1 <t-base User-space Components

The usual way to build t-base user space components is from a complete Android source tree:

Simply copy the *mobicore* folder to an Android source tree; all the components embed their own internal *makefile.mk*.

You should have the following organization:

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

They will be automatically built within the next build of full Android source tree.

Else usual Android command can be used to build only these components:

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

2.1.2 <t-base Kernel-space Components

The <t-base Linux driver directory in the <t-base package is designed to be directly dropped in the Linux kernel source tree, in the *drivers* folder.

It contains two modules:

- ◀ The <t-base Linux driver (with two interfaces for applications and <t-base Daemon).
- ◀ The <t-base Kernel API.

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

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

Please note that dropping it in a different folder in the Linux source tree will not work.

To enable automatic building inside the kernel follow these steps:

- ◀ Update Linux kernel configuration file *linux/drivers/Kconfig* with following line (must be inserted before the last *endmenu* line):

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

- ◀ And add the <t-base Linux driver to the Linux kernel build file *linux/drivers/Makefile* :

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

- ◀ Then, run *make menuconfig* and in the *Device Drivers* page of the configuration menu, please select:

- ◀ *Linux Mobicore Support*
- ◀ *Linux Mobicore API*

```
<*> Linux Mobicore Support
[ ]   Mobicore Module debug mode
<*>   Linux Mobicore API
```

- ◀ Finally, run *make* again and the <t-base 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 table below gives the directories in the Android file system where <t-base components should be placed:

2.2.1.1 32 bit Android OS file system:

Component	Name of the binary	Device directory
<t-base Daemon	mcDriverDaemon	/system/bin/
Root Provisioning Agent	RootPA.apk	/system/app/
<t-base Client library	libMcClient.so	/system/lib/
<t-base registry interface	libMcRegistry.so	/system/lib/
Keymaster library	libMcTeeKeymaster.so	/system/lib/
Curl library	libcurl.so	/system/lib/
Log library	libcommonpawrapper.so	/system/lib/
Content Management Trusted Application	07010000...0000.tlbin (tlcm.axf)	/system/app/mcRegistry
Secure Storage Driver	07050500...0000.tlbin (DrSecureStorage.axf)	/system/app/mcRegistry

2.2.1.1 64 bit Android OS file system:

Component	Name of the binary	Device directory
<t-base Daemon 64 bit	mcDriverDaemon	/system/bin/
Root Provisioning Agent 32 bit	RootPA.apk	/system/app/
<t-base Client library 32 and 64 bit	libMcClient.so	/system/lib/ && /system/lib64/
<t-base registry interface 32 and 64 bit	libMcRegistry.so	/system/lib/ && /system/lib64/
Keymaster library 32 and 64 bit	libMcTeeKeymaster.so	/system/lib/ && /system/lib64/
Curl library 32 bit	libcurl.so	/system/lib/
Log library 32 bit	libcommonpawrapper.so	/system/lib/

Content Management Trusted Application	07010000...0000.tlbin (tlcm.axf)	/system/app/mcRegistry
Secure Storage Driver	07050500...0000.tlbin (DrSecureStorage.axf)	/system/app/mcRegistry

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

If the <t-base Linux driver modules are not built-in the Linux kernel, then you should have the following:

Component	Name of the binary	Device directory
<t-base user device	mcDrvModule.ko	/system/lib/modules/
<t-base kernel device	mcKernelApi.ko	/system/lib/modules/

2.2.1 Permissions and Access Rights

- ◀ <t-base daemon:
When using Android integration Daemon should automatically get the correct permissions (0755)
- ◀ <t-base libraries:
No specific permission is needed for libraries.
- ◀ <t-base linux driver:
<t-base linux driver has two devices, device for user access and device for daemon access (system).

Access permissions for these are defined by *udev* following way:

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

2.2.2 Directories Requirements

There are two directories that should be created in device file-system for <t-base. These paths are used by <t-base components at runtime.

2.2.2.1 mcRegistry directory

This directory is used by t-base to store non-permanent contents, that can be deleted with factory reset or device wipe. Usually it used to store third party trustlet content (OTA installed containers).

Mandatory path is:

```
/data/app/mcRegistry
```

and permissions should be 771 (System, Read/Write/Execute; User list).

2.2.2.2 Persistent Trustlets and Secure Drivers

Trustlets and Secure Drivers can be stored anywhere in the device file system. However, Persistent Trustlets or Secure Drivers binaries must be placed in a persistent partition of file system, persistent among factory reset or device wipe.

For legacy reason, some System Trustlets (like the tlcm, Content Management System Trustlet) are expected to be found in

```
/system/app/mcRegistry
```

(No content generated here at runtime by t-base, only used to store legacy persistent System Trustlets)

2.2.3 Starting up <t-base

For automatically running <t-base daemon at start-up, add the following lines to the *init.rc* file of the Android device:

```
service mobicore /system/bin/mcDriverDaemon
    user system
    class main
```

Daemon has to have system permissions for user and group. Otherwise OEM is responsible to set correct parameters for the service.

OEM is also responsible to decide how the system should behave in error cases like if the start-up fails or if the daemon crashes.

3 <T-BASE SECURE WORLD COMPONENTS

Secure World components of <t-base are available in the package directory *SecureWorldIntegration*. It contains:

- ◀ The <t-base binary (secure operating system running in Secure World).
- ◀ The Content Management trusted application (System trusted application responsible for Containers management).
- ◀ Mandatories Secure World drivers used by <t-base itself.

The <t-base images are provided in binary (Debug or Release) under *SecureIntegration/t-base/bin/MobiCore/*:

- ◀ the Debug version is much slower but prints all <t-base traces to the Linux Kernel log(dmesg)
- ◀ the Release version is very fast but there are not internal <t-base traces (note that even if you use the Release image of <t-base you can get traces from the trusted applications if you have compiled them 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 package contains also images signed with Trustonic test keys (*.img*).

The Content Management trusted application is provided in binary (Debug and Release) under *SecureIntegration/tbase/bin/TLcm*. The trusted application is provided un-configured (*tlcm.axf*) and should be configured with the <t-base KPH request signing key and signed with the System trusted application key pair.

3.1 <T-BASE IN BOOT CHAIN

<t-base image should be started up in the boot flow as early as possible. Booting up with <t-base is a critical task in the system design and it should be done in co-operation with SOC vendor and Trustonic. There are several options for implementing <t-base boot. The purpose of this chapter is to give basic level of understanding what needs to be done.

- ✧ It is recommended to store <t-base to a permanent location in Boot partition or some other permanent storage. Usually early in the bootchain there is no normal file system present at that time.
- ✧ <t-base image must be signed. The responsibility of doing this is usually on OEM/ODM.
- ✧ <t-base image should be loaded to secure memory (internal or DRAM). Loading of <t-base image to secure memory should be done by bootloader or by Uboot.
- ✧ After loading <t-base into secure memory signature of <t-base image should be verified.
- ✧ Before calling <t-base, the caller needs to set boot configuration block for <t-base. This block is used for exchanging some predefined data between the caller and <t-base.
- ✧ Calling <t-base can usually be just a basic jump instruction. Depending on the execution environment, something else might also be needed to be done. As the initialization of <t-base returns to the caller, it means that the caller environment (bootloader/uboot) should be saved and restored for the time period execution is in TrustZone side.

If <t-base signature check or boot fails, it is up to OEM or SOC vendor to decide whether the device can be booted without <t-base or not.

3.2 TRUSTED APPLICATIONS AND SECURE DRIVERS OVERVIEW

In <t-base environment, there are 3 types of secure components:

- < System trusted applications,
- < Service Provider trusted applications,
- < Drivers.

Service Provider trusted applications are applications deployed at runtime over the air. Service Provider trusted applications are deployed through a Trusted Service Manager (TSM) which is not covered by this document.

System trusted applications and drivers are “pre-installed” on the device. To ensure the security of these trusted applications or drivers, they need to be signed and verified at runtime by <t-base. To do this, the hash of the System trusted applications public key is inserted to <t-base raw-image.

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

Any UUID's can be used by OEMs. There is no acceptance control of UUID's done by Trustonic. However, UUID's having 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 applications and drivers development, please check documentations provided in <t-sdk directories.

3.3 CONFIG AND SIGNING OF T-BASE COMPONENTS

The following section describes how to manage the different private and public keys in < t-base product to ensure the authenticity and integrity of < t-base binary and System trusted applications.

It is mandatory for the OEM to go through all the steps of this section to customize the keys involved in the different security mechanism (for testing purpose, the < t-base integrator would also have to do these steps).

3.3.1 Keys in < t-base Environment

There are four keys involved in t-base environment that are needed to be taken care of in the integration phase:

- ✧ *PrK.Vendor.Boot* and *PuK.Vendor.Boot*:
< t-base signing PKI key pair for boot, these keys are used for verifying integrity of < t-base image.
- ✧ *PrK.Vendor.TltSig* and *PuK.Vendor.TltSig*:
< t-base System trusted applications signing PKI key pair, they are used for signing System trusted applications and drivers that are installed by default on the device.
- ✧ *PuK.Trustonic.Endorsement*
The < t-base endorsement key is used by Trusted Applications to sign data and prove that it is generated in a genuine TEE and in the right Trusted Application.
This key is directly provided by Trustonic in < t-base- package.
- ✧ *Prk.Kph.Request* and *PuK.Kph.Request*:
The < t-base KPH request signing key pair, it is used to verify the setting up of < t-base Content Management in production (used to verify exchanges with KPH).
This key, only involved during productions, **is directly shared by Trustonic with the OEM** (covered by documentation *t-base_Provisioning_Manual.pdf*).

OpenSSL can be used to generate the key pairs that will be injected or used with < t-base components:

```
$ 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 < t-base components).

3.3.2 Configuring <t-base Binary

3.3.2.1 Configuring <t-base Image

The first step to configure <t-base image is to inject the hash of the <t-base System trusted application signing public key (*PuK.Vendor.TltSig*) into the raw <t-base image. The goal is to ensure origin of System trusted applications and drivers.

The second step is to inject the endorsement key. This key is provided by Trustonic. It is located in "*SecureIntegration\t-base\bin\MobiCore\endorsementPubKey.pem*". Please refer to the <t-base Developer's Guide for details about the Endorsement API.

The MobiConfig tool is provided in t-base package to inject keys into a binary (*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
```

3.3.2.2 Signing the System Trusted Applications and Drivers

Then, all the System trusted applications and drivers of the system must be signed with the System trusted application Signing key.

The MobiConvert tool is provided in t-base package to sign a trusted application (*t-base-dev-kit/t-sdk/TlSdk/Bin/MobiConvert*, also details in <t-sdk documentations):

```
$ 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
```

3.3.2.3 Signing <t-base Image

Signing <t-base, with *VendorBoot.pem*, has to be done by OEM methods and signature check of <t-base image, before starting up, has to be ensured by the platform bootloader.

3.3.3 Configuring Content Management Trusted Application

The Content Management trusted application is the secure peer of the Normal World t-base 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).

3.3.3.1 Configure 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 <t-base 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
```

3.3.3.2 Signing 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 finally be installed on the device in the Registry (the persistent one, in case of device wipe, the Content Management must still work).

3.3.4 Signing the Secure Storage Driver

The Secure Storage driver is a new driver coming with <t-base-300 which is used for GP Secure Storage and which 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 drSecureStorage.axf
  -output 07050500000000000000000000000000.drbin
  -d 259
  -memtype 2
  -flags 0
  -interfaceversion 1.0
  -keyfile VendorTltSig.pem
$ cp 07050500000000000000000000000000.drbin
07050500000000000000000000000000.tlbin
```

The signed Secure Storage driver must finally be installed on the device in the Registry (the persistent one, in case of device wipe, the GP trusted Applications must still work).

3.3.5 Signing the Trusted User Interface Driver

The Trusted User Interface driver is a new driver coming with <t-base-300 which 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 4
  -bin drTui.axf
  -output 070c0000000000000000000000000000.drbin
  -d 257
  -memtype 2
  -flags 0
  -interfaceversion 1.0
  -keyfile VendorTltSig.pem
$ cp 070c0000000000000000000000000000.drbin
070c0000000000000000000000000000.tlbin
```

The signed driver must finally be installed on the device in the Registry (the persistent one).

3.3.6 Signing the DRM Driver

The DRM driver is a new driver coming with <t-base-300 which 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 070b0000000000000000000000000000.drbin
    -d 1536
    -memtype 2
    -flags 0
    -interfaceversion 1.0
    -keyfile VendorTltSig.pem
$ cp 070b0000000000000000000000000000.drbin
070b0000000000000000000000000000.tlbin
```

The signed driver must finally be installed on the device in the Registry (the persistent one).

3.3.1 Signing the TEE Keymaster Trusted Application

The TEE Keymaster Trusted Application is a new component coming with <code>t-base-301</code> 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 signed Tee Keymaster Trusted Application must be installed on the device in the persistent Registry (The TEE Keymaster must still work after factory reset or device wipe).

3.4 <T-BASE BOOT PARAMETERS ON 32 BIT HARDWARE

The entity that starts up <t-base (bootloader or Uboot) should use <t-base boot configuration block as an interface towards <t-base. The configuration block can be customized according to needs. It should transfer at least the following information to <t-base.

Example of <t-base boot args 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. <t-base jumps to this NWD address after changing the NS bit.

Here is the definition of the <t-base 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;
```

3.5 <T-BASE BOOT PARAMETERS ON 64 BIT HARDWARE

On this type of hardware platform, the entity that starts up <t-base is the *Trusted Firmware* (reference version provided by ARM). To start a new integration, Trustonic will provide a reference version of it, with included an implementation of the *Secure Dispatcher* in charge of the <t-base component.

For reference, here is the definition of the <t-base 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 t-base comm.
    uint32_t RFU2;
    uint64_t flags;
} bootCfg_t;
```

3.6 FASTCALLS HOOK MECHANISM

A FastCall is a call to the ARM SMC instruction with some specific parameters which allows executing some routines in <t-base without performing a complete context switch.

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

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)

<t-base already supports either generic or platform-specific internal FastCalls. They are mainly used for <t-base initialization and common operations that have to be done by the normal world but can only be performed in secure.

FastCalls have the following limitations:

- ✧ They cannot call any TIApi 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 in a FastCall.

3.6.1 Handling FastCalls

<t-base 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 <t-base. "Firmware Driver" is a naming convention for the first two drivers to ever install a FastCall handler during <t-base 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 FastCall related to the silicon platform and the second Firmware Driver is intended to be developed by the OEM to handle FastCall 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 t-base 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 he is handling.

Regarding the information shared between the Firmware driver and the Fastcall handlers: the Fastcall handlers get Firmware driver memory mappings in range of 0-2MB at the time handler is installed. As there is absolutely no synchronization mechanism between the Firmware driver and the FastCall handles once installed, it's 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.

3.6.2 Additional Secure Driver APIs

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

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

3.6.2.1 Types

3.6.2.1.1 FastCall Registers

```
typedef word_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 catch by the Firmware driver, they are forwarded by <t-base to the Firmware driver through the `fastcall_registers_t` table. The size of this table can be checked by field `registers` from FastCall context structure (`struct fcContext`)

3.6.2.1.2 FastCall Context

```
struct fcContext {

    /* Size of the context */
    word_t size;

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

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

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

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

The FastCall context structure, `fcContext_t`, is filled when the Firmware driver is initialized into <t-base and forwarded as 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 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 <t-base version values from 0 to 7 are valid.
- < `entry`: The L1 descriptor that will be used for the mapping.

Value returned by function `setL1Entry` is the virtual address of the mapped area (Null in case of error).

However, these mappings, done in 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 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 one to one mapping area.
- < `length`: The size of one to one mapping area.
- < `flags`: Currently not in use.

The address range [0x0000 0000 - 0x01ff ffff] is today reserved for <t-base internal usage. In consequence, the range passed in parameter for the one to one mapping must not overlap this area (else, mapping in <t-base range will be ignored).

Note: This feature is highly system dependent and may not be available as is in all <t-base versions.

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).

Note: This feature is highly system dependent and may not be available as is in all <t-base versions.

3.6.2.2 Specific FastCall Entry Points

3.6.2.2.1 FastCall Handler Initialization

```
typedef word_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 goes OK. Else, with any other value, the fastcalls will not be allowed.

3.6.2.2.2 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. It is executed 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. 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

It should return non zero when a fastcall is handled internally and zero when the FastCall is not known.

3.6.2.3 Specific Firmware Driver APIs

3.6.2.3.1 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 3.6.3, "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

3.6.3 Firmware Driver Structure

3.6.3.1 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);
}
```

3.6.3.2 Assembly Glue for 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 on 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 call 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

```

3.6.3.3 FastCall Handler Example

The FastCall Handler is divided in two steps:

- ◀ The initialization function, called by <t-base when the Firmware driver install the FastCall hook:

```

word_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 <t-base 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;
}
```

3.6.4 u-boot Integration Sample

This section describes what could be the integration of the Firmware driver in u-boot, and directly load it into <t-base.

The advantage would be to have the FastCall hook available for u-boot and for the very early boot sequence of the Normal World OS.

3.6.4.1 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

NOTE: This guide assumes the bootloader developer has a mechanism to load the secure driver from permanent storage to memory AND has done so before integrating.

NOTE: It is assumed that if TCI is required the bootloader will not release that particular memory for other uses

3.6.4.2 <t-base Communication Setup

The function `mci_setup()` handles the setup of the communication mechanism between the bootloader and <t-base

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 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];
```

NOTE: This buffer must be globally defined as it is used for communication throughout the code!

NOTE: This code execute several smc calls to give <t-base 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!

3.6.4.3 Firmware Driver Loading

The function for driver loading 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

NOTE: As stated before please note that if the secure driver will use the TCI after initialization you MUST ensure the memory allocated for TCI is not released to Android!

NOTE: Please remember 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 <t-base might not relinquish control to bootloader at all there is nothing the code can do!

3.6.4.4 Data Deallocation

Data deallocation is very important in the bootflow. Without proper deallocation subsequent calls to <t-base from the Android daemon will fail.

The code 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 uninitialized\n");
        return -1;
    }
    return 0;
}

```

NOTE: Error handling is important here as any error returned by this code will result in an unusable <t-base system for Android

4 TEE KEYMASTER

<t-base TEE keymaster feature provides the support of all the cryptographic algorithms required to be compliant with the Google Android Keymaster.

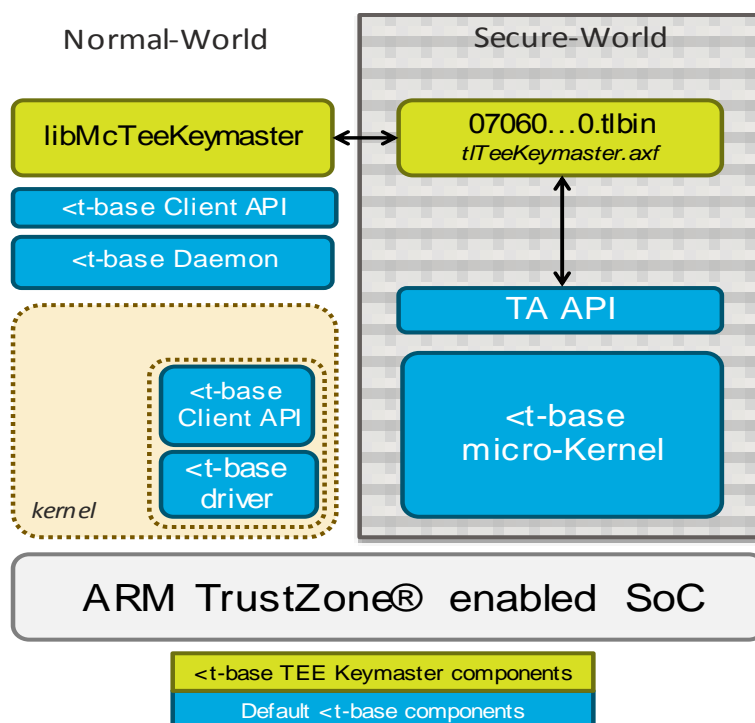
The following algorithms are supported:

- ✧ RSA (both normal and CRT) up to 4096 bit key sizes. Raw RSA operations are 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:



4.1 NORMAL WORLD CONNECTOR AND INTEGRATION

The <t-base package contains the Normal-World library in charge of doing the connection between the Google Android keymaster framework and <t-base secure OS.

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

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

The header file ,tlcTeeKeymaster_if.h, that define the interfaces to communicate with the <t-base is available here:

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

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

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

The HAL keystore module in Linux kernel is not included in this library and his not part of <t-base product. However, as a generic statement, keymaster HAL module development would involve the following steps:

- ◀ Implement HAL_MODULE_INFO_SYM and assign correct keymaster functions in xxxxx_km_open APIs.

```
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 keymaster APIs (generate_, import_, get_keypair_, sign_, verify_) with calls to <t-base TEE keymaster APIs detailed in interface file tlcTeeKeymaster_if.h.

- Finally update `Android.mk` to generate file `keystore.xxxx.so` (which should be placed on the device in the following directory `/system/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)

```

4.2 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 <t-base, where the different cryptographic algorithms are implemented.

The source and the binary of the Trusted Application are available in <t-base package under:

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

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

5 DRM INTEGRATION

<t-base 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

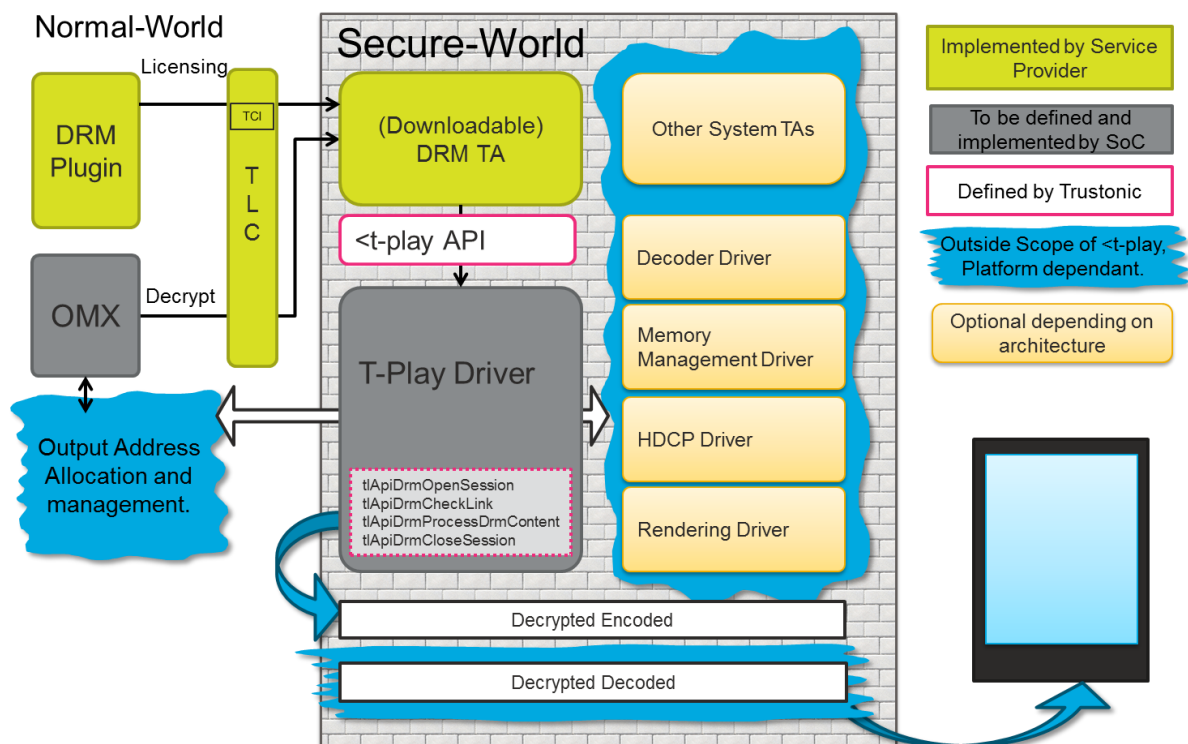


Figure 1: DRM Components Overview

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 <t-play 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 <t-play 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 <t-play 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 <t-base 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 map 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 <t-base 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 <t-play Secure Driver UUID is :

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

Secure Playback driver ID

The <t-play 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;
        tIDrmApiDrmContent_t  drmContent;
        tIDrmApiLink_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 output 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 <t-base 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 <t-base implements an isolation between trusted applications. Each trusted application executes in its own memory space and any persistent 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

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

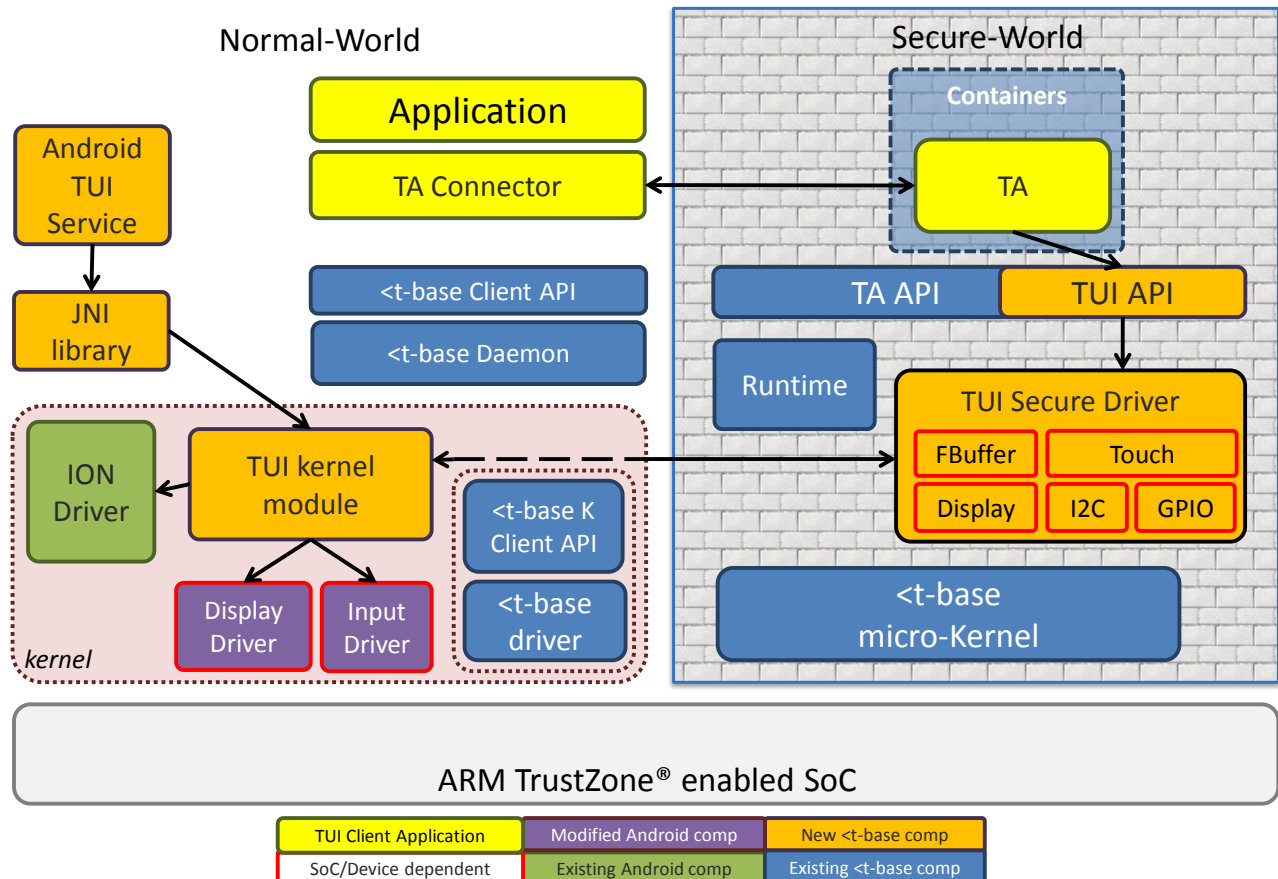


Figure 2: TUI Components Overview.

- ✧ 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 module running in the normal world.
- ✧ The TUI kernel module is a proxy that links the TUI secure driver to the UI Linux drivers and a TUI Android service.
- ✧ The UI Linux drivers – typically touch, i2c and GPIO drivers for input, and display drivers – 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 Android service is designed to notice them and trigger the cancellation of the TUI session.

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, <t-base 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 framebuffer, and optionally the display controller.

In case the framebuffer is allocated from secure world memory, the secure bootloader is responsible for setting up the TZASC and the framebuffer 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 framebuffer is allocated from normal world memory, it 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 buffer, therefore the secure bootloader must wipe the secure framebuffer from the previous boot before any normal world component is executed.

In case the framebuffer is 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 framebuffer at a fixed address.

If the framebuffer is 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 framebuffer is 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 HAL function `tuiHalTouchReset` needs to perform the following operations:

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

The TUI driver requires twice 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 only uses one framebuffer 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) 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 given as an answer to the command `CMD_TUI_SW_OPEN_SESSION`.

6.2.2 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:

- < `tuiHalFBOpen`:
 - < It is called at least twice, one for the working buffer, one for the framebuffer.
 - < 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.
- < `tuiHalDisplayInitialize`: initialize the display system for the Secure World.
- < `tuiHalFBProtect`: 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 point is called within TUI session:

- < `tuiHalFBImageBlt`: copy an image to the actual 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).

The following entry points are called at TUI session closing:

- < `tuiHalFBUnProtect`: 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.

6.2.3 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:

- ◀ `tuiHalTouchGetInfo`: called from the core driver main thread at session opening.
- ◀ `tuiHalTouchOpen`: called from the core driver main thread after `tuiHalTouchGetInfo` 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.
- ◀ `tuiHalTouchClose`: called from core driver main thread at TUI session closing just before killing the touch thread. Before returning this function must 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:

- ◀ `drTouchReport`: called from the touch thread to signal a touch event to the core driver.
- ◀ `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.

6.2.4 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
```

6.2.5 Integrating the TUI secure driver

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

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

6.3 TUI KERNEL COMPONENTS

6.3.1 TUI module

The TUI kernel module 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 module driver loading:

- < `uint32_t hal_tui_init(void)`
 This function is called when the kernel module is initialized, either at boot time, if the module is built statically in the kernel, or when the kernel is dynamically loaded if the module is 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 kernel module exit. It is called when the kernel module is unloaded, for a dynamic kernel module, and never called for a module built 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.

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.
 - ◀ `CMD_TUI_SW_STOP_DISPLAY`:
 - ◀ 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.
- ◀ 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 <t-base Kernel-space Components.

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

The <t-base TUI module can be found in `AndroidIntegration/Src/tui/TlcTui`. The contents of `TlcTui/Locals/Code/` must be copied to `<linux>/drivers/gud/t-base-tui/`.

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 "<t-base TUI"
    depends on MOBICORE_API
    ---help---
        Enable <t-base Trusted User Interface

config TRUSTONIC_TRUSTED_UI_FB_BLANK
    bool "<t-base TUI with fb_blank"
    depends on TRUSTONIC_TRUSTED_UI
    ---help---
        Blank the framebuffer before starting a TUI session
```

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

```
obj-$(CONFIG_TRUSTONIC_TRUSTED_UI) += mcDrvTui.o

mcDrvTui-objs := t-base-tui/main.o \
                t-base-tui/tlcTui.o \
                t-base-tui/trustedui.o

ccflags-$(CONFIG_TRUSTONIC_TRUSTED_UI) += -Idrivers/gud/t-base-
tui
ccflags-$(CONFIG_TRUSTONIC_TRUSTED_UI) += -
Idrivers/gud/MobiCoreKernelApi/public
ccflags-$(CONFIG_TRUSTONIC_TRUSTED_UI) += -
Idrivers/gud/MobiCoreKernelApi/include

ccflags-$(CONFIG_MOBICORE_DEBUG) += -DDEBUG
ccflags-$(CONFIG_MOBICORE_VERBOSE) += -DDEBUG_VERBOSE
```

- < Then, run `make menuconfig` and in the *Device Drivers* page of the configuration menu, please select:

- < *Mobicore Driver*
- < *Linux Mobicore API*
- < *<t-base-TUI*
- < *<t-base TUI with fb_blank*

```
<*> Mobicore Driver
[ ]   Mobicore Module debug mode
<*>   Linux Mobicore API
<*>   <t-base TUI
<*>   <t-base TUI with fb_blank
```

- < Finally, run `make` again and the <t-base TUI kernel components will be included in the kernel image.

6.3.4 Integrating the TUI module

The TUI module 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 `OnStartCommand()`
- < 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 t-base TUI Android component is from a complete Android source tree:

Simply copy the contents of the folder `AndroidIntegration/Src/tui/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 <t-base 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/
```


7 RUNNING TESTS

The <t-base product comes with a basic test suite which contains tests to validate the <t-base integration on the platform.

The test suite is located in the `Test-Package-SIP` folder.

The test suite contains complete documentation explaining how to run the tests.