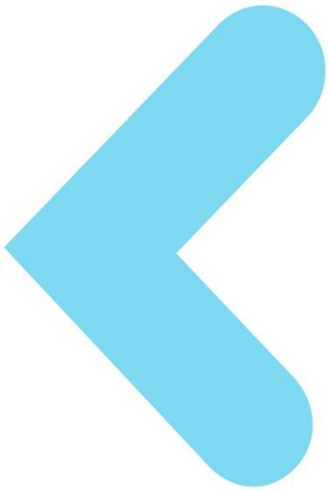
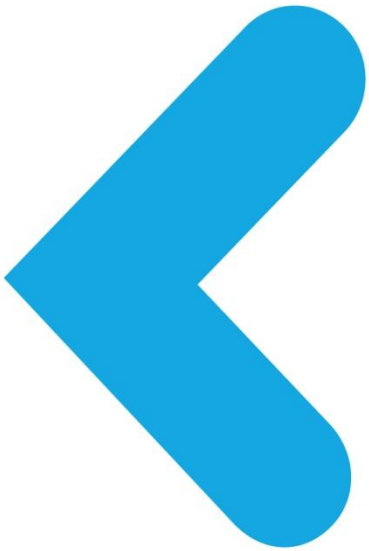
TRUSTONIC



Trusted Application Developer's Guide

<t-sdk



PREFACE

VERSION HISTORY

Version	Date	Modification
1.0	February 6th, 2013	First Issued version
2.0	March 21st , 2013	Several documentation enhancements
3.0	April 30th, 2013	Added support for GNU GCC toolchain
4.0	July 5th, 2013	Several documentation enhancements
5.0	July 31st, 2013	Added Debug Agent paragraph. Added Documentation for Windows Client Applications. Documentation improvement.

TABLE OF CONTENTS

1 INTRODUCTION

This guide is a practical introduction for developing:

- <t-base Trusted Application
- a corresponding Trusted Application Connector for Android

It explores the concepts behind <t-base, the framework for constructing a security aware application, and the tools for developing software for the platform.

The Developers Guide contains a subset of the documentation for the <t-base platform related to application development, for the complete reference, refer to [API].

1.1 REFERENCED DOCUMENTS

API	<t-base API Documentation

1.2 GLOSSARY AND ABBREVIATIONS

ADB	**A**ndroid **D**ebug **B**ridge
API	**A**pplication **P**rogramming **I**nterface
BSS	The BSS segment contains all uninitialized data.
DMA	**D**irect **M**emory **A**ccess
DS-5™	ARM **D**evelopment **S**tudio 5
GDB	**G**nu **D**e**B**ugger
GNU	**G**NU's **N**ot **U**nix
IPC	**I**nter-**P**rocess **C**ommunication, communication between two tasks running in different processes.
JNI	**J**ava **N**ative **I**nterface
JTAG	**J**oint **T**est **A**ction **G**roup
MMU	**M**emory **M**apping **U**nit
NDK	Android **N**ative **D**evelopment **K**it
OS	**O**perating **S**ystem
OTA	**O**ver-**T**he-**A**ir

PID	**P**rocess **ID**
RFC	**R**equest **F**or **C**omments published by the Internet Engineering Task Force (IETF)
RPC	**R**emote **P**rocedure **C**all
<t-sdk	<t-base **S**oftware **D**evelopment **K**it
SP-TA	Service Provider Trusted Application - Trusted Applications developed by Application developers and not OEM installed System Trusted applications
TCI	**T**rusted Application **C**ontrol **I**nterface
TEE	Trusted Execution Environment. <t-base is an implementation of a TEE. The TEE is standardized by GlobalPlatform.
TA	**T**rusted **A**pplication on the TEE executing specific security services in the <t-base runtime environment.
TLC	**T**rusted Application **L**ayer **C**onnector. Software running in the Normal-World providing a high level convenience interface to access a Trusted Application in the Secure-World by the Normal-World client.
TZ	ARM TrustZone
UART	Universal Asynchronous Receiver/Transmitter
UUID	Universal Unique Identifier. The UUID is an identifier standard used in software construction, standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). The UUID in the <t-base ecosystem is used to uniquely identify Trusted Applications. See RFC 4122.
WSM	World shared memory. Shared memory which will be used for IPC between Normal-World and Secure-World.

2 <T-BASE PRODUCT OVERVIEW

<t-base is a portable and open Trusted Execution Environment (TEE) aiming at executing Trusted Applications on a device. It includes also built-in features like cryptography or secure objects. It is a versatile environment that can be integrated on different System on Chip (SoC) supporting the ARM TrustZone technology.

<t-base uses ARM TrustZone to separate the platform into two distinct areas, the Normal-World with a conventional rich operation system and rich applications and the Secure-World.

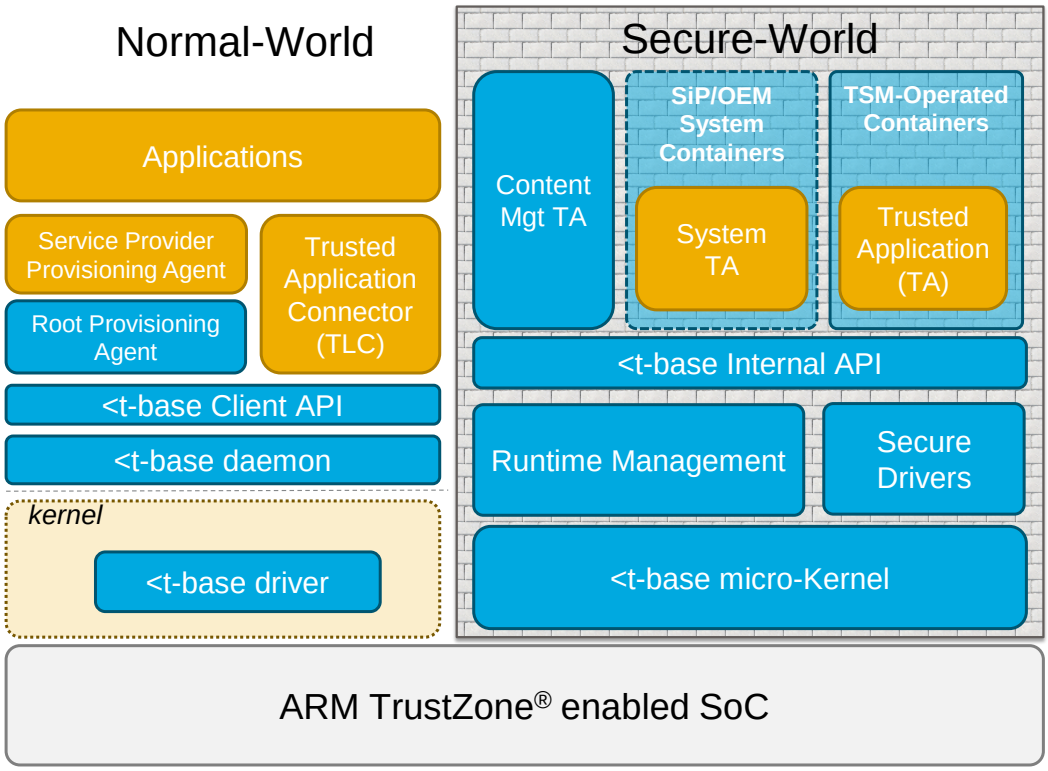


Figure 1: <t-base Architecture Overview.

The Secure-World contains essentially the <t-base core operating system and the Trusted Applications. It provides security functionality to the Normal-World with an on-device client-server architecture. The Normal-World contains mainly software which is not security sensitive (the sensitive code should be migrated to the Secure-World) and it calls the Secure-World to get security functionality via a communication mechanism and several APIs provided by <t-base. The caller in the Normal-World is usually an application, also called a client.

The Trusted Applications in the Secure-World are installed in Containers. A Container is a security domain which can host several Trusted Applications controlled by a third party. There are two kinds of Containers:

- The TSM-Operated Containers, which are created at runtime under the control of Trustonic. Trusted Applications of a TSM-Operated Container can be administrated Over-The-Air via a Trusted Service Manager (TSM).
- The System Container, which is pre-installed at the time of manufacture along with some Trusted Applications. Trusted Applications in the system container cannot be downloaded or updated Over-The-Air via a TSM.

The root Provisioning Agent is a Normal-World component which communicates between the Device and TRUSTONIC's backend system to create TSM-Operated Containers within <t-base at runtime.

Note that in order to enable <t-base and the TSM-Operated Containers on a Device, the OEM must install Trustonic's Key Provisioning Host (KPH) at its manufacturing line. The KPH is a tool which injects a key on the device and which stores a copy in the Trustonic backend system.

<t-base provides APIs in the Normal-World and the Secure-World. In the Normal-World, the <t-base Client API is the API to communicate from the Normal-World to the Secure-World. This API enables to establish a communication channel with the Secure-World and send commands to the Trusted Applications. It enables also to exchange some memory buffers between the Normal-World and the Trusted Applications.

In the Secure-World, <t-base provides the <t-base Internal API which is the interface to be used for the development of Trusted Applications.

Furthermore, the <t-base architecture supports Secure Drivers to interact with any secure peripherals.

3 DESIGN PRINCIPLES

An operating system cannot provide the trustworthiness that is needed to keep up with the ongoing improvements of attacks and threats, whilst continually adding new functionalities alone. This means that security mechanisms need to be integrated in the system at the lowest possible level instead of being added on top. <t-base is able to provide such a secure environment to protect security critical data and applications against attacks with the use of TrustZone.

The following chapter describes what is needed to fully take advantage of the functionality <t-base provides for an application developer.

3.1 SECURE OR NOT SECURE?

First, one needs to identify and define the security critical parts of the application.

The prerequisite for implementing a Trusted Application and TLC is to determine the parts which have to be incorporated within a Trusted Application. The aim should be to keep the Trusted Application part as small as possible. The main reason is that a big code base has potentially a lot of bugs; or, in other words, you should keep the system simple to have it working properly and secure.

In the following, a guideline is given for keeping the Trusted Application as small as possible:

1. Determine sensitive/critical information
2. Identify the parts that deal with this information
3. Isolate these parts in a Trusted Application

For example, in the payment Use Case, the safety critical information would be the amount of money to be transferred and the participants of the transaction. Furthermore, the user has to enter PIN and TAN to identify himself to get access to the banks service and sign a transaction. All the software components handling this sensitive information need to be identified. On the client side, this would mainly be the user interface, as it collects the information from the user. Therefore, the PIN and TAN entry need to be controlled by a Trusted Application which can use a secure driver, making it impossible for the Normal-World to eavesdrop on the user secrets.

3.2 TRUSTED APPLICATION CONNECTOR DESIGN GUIDELINES

The Trusted Application Connector (TLC) in the Normal World is the counterpart to the Trusted Application in the Secure World. The TLC is responsible for establishing a connection with the Trusted Application and for providing the Trusted Application's features to the upper layers, e.g. an Android App.

It uses the <t-base Client API to initiate starting and stopping of a Trusted Application and to exchange data with it. To the upper layer, e.g. the Android application, the TLC makes the Trusted Application's security features accessible. Therefore the TLC developer designs an API for the Normal-World application that corresponds to these security features. Android applications then use this Trusted Application via its specific TLC API.

A Trusted Application Connector can be viewed as a library for the functionality provided by one or more Trusted Applications. This involves:

- loading and opening a session to one or more Trusted Applications
- opening and closing sessions to one or more Trusted Applications
- marshalling function calls to a Trusted Application via TCI buffer and waiting for the Trusted Application to respond
- mapping and un-mapping additional memory to a Trusted Application session
- handling Trusted Application return codes
- closing sessions with the Trusted Applications

3.3 TRUSTED APPLICATION DESIGN CONSIDERATIONS

When developing a Trusted Application, keep in mind that it will be running in a resource constrained environment. While restricting the access of your Trusted Application, <t-base will also protect your Trusted Application from other Trusted Applications.

3.4 TRUSTED APPLICATION ADDRESS SPACE

The Trusted Application address space is divided by <t-base in a secure section and an area of World Shared Memory (WSM). WSM allows the Trusted Application to communicate with the normal world.

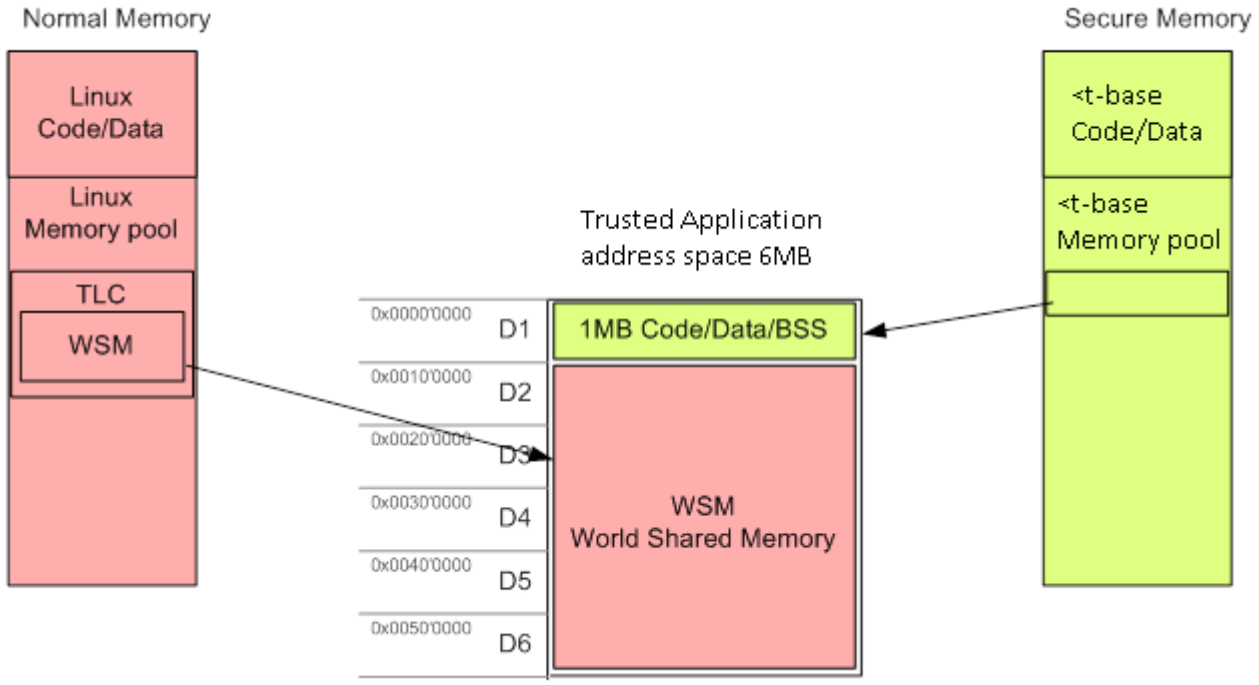


Figure 2: Secure areas of Trusted Application Address Space.

The virtual address space of the Trusted Application has following characteristics:

- Limited to 6 MB of virtual address space.
- Divided into 6 "D" regions of 1 MB each.
- Only region D1 is secure memory.

- Region D2 is shared memory set up during mcOpenSession() call.
- Regions D3 to D6 are shared memory set up by mcMap() call.

Your Trusted Application code segment and the combined data and BSS segments including stack are allocated by <t-base in secure memory and put into region D1. The total size of your Trusted Application is limited to 512k.

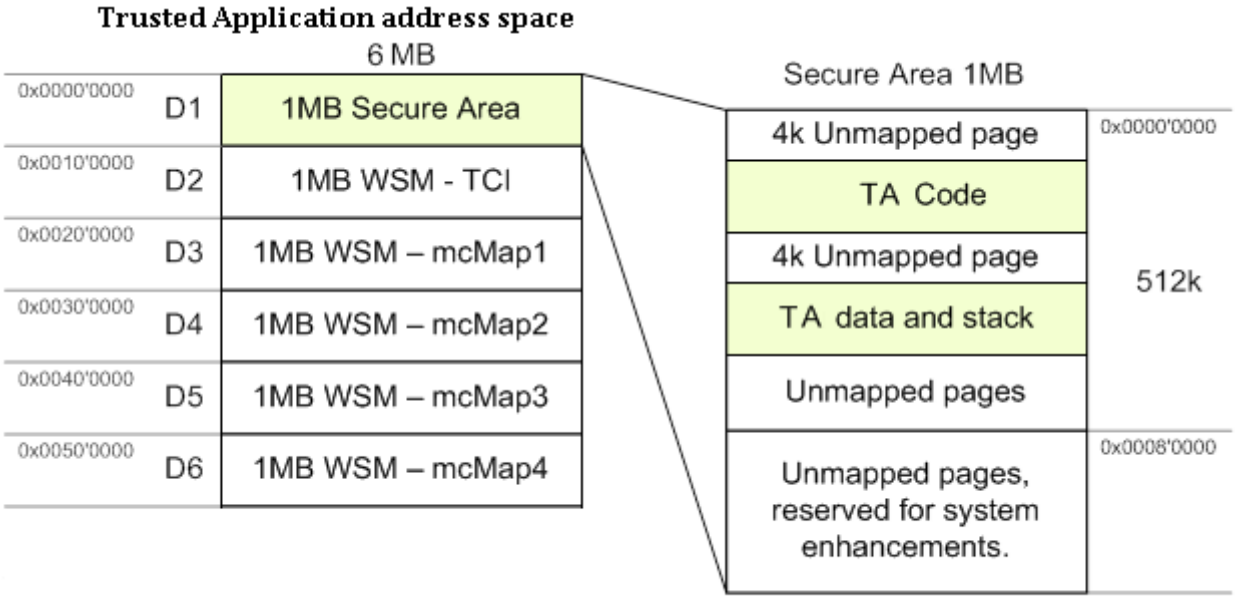


Figure 3: Trusted Application Address Space.

<t-base does not provide demand paging. You cannot request more secure memory at runtime of your Trusted Application.

- This means that you cannot use malloc() or alloca() to reserve memory on heap or stack.
  Use static buffers instead (e.g. global variables) to store temporary data.

3.4.1 Trusted Application interactions with the rest of the system

Following limitations ensure the security of your Trusted Application:

- Follow an event driven design. Separate your Trusted Applications functionality into multiple methods, so the Trusted Application is able to return fast.
- Keep in mind that the Trusted Application execution may be interrupted by the Normal-World anytime (e.g. due to an incoming phone call).
- A Trusted Application process is limited to a single thread. In case you need to have multiple threads you need to create separate Trusted Applications.
- Trusted Applications are not able to communicate directly with each other. But they can share Secure Objects to exchange data.
- Trusted Applications do not have direct hardware access, only with the use of the Trusted Application API.

- Try to avoid busy waiting and polling to conserve battery life.
- Never write security critical data to the TCI unencrypted. Your Trusted Application may get interrupted at any time, before being able to clean the TCI buffer, and the Normal-World will be able to read the TCI content.

The result of your Trusted Application design should be an API of one or more function calls to your Trusted Application.

3.5 INTER-WORLD COMMUNICATION

The following sections describe the functions necessary for the communication between the TLC and the Trusted Application via Inter-World communication.

3.5.1 The Basics

The term Inter-World Communication used within this document is very similar to Inter-Process Communication (IPC), with the difference that the two tasks are not only running in different processes but also in different "worlds", the Normal-World and the Secure-World.

IPC defines methods for data exchange between tasks that by definition work in separated address spaces. The intentional consequence is, that these tasks cannot communicate between each other unless through a particular communication mechanism.

3.5.2 Communication Mechanisms

IPC technology makes a difference between

- message passing
- synchronization
- shared memory
- remote procedure calls (RPC)

The type of communication mechanisms used, very often depends on the underlying hardware, e.g. DMA controllers, addressing mechanisms and MMU capabilities. Also the bandwidth and amount of data to be transferred between tasks is relevant. For example, a few bytes can easily be transferred through processor registers whereas a video stream requires quite a different approach when streaming data between tasks.

3.5.2.1 <t-base Inter-World Communication

For <t-base, communication between the Normal-World and the Secure-World is facilitated by writing data to a world shared memory buffer. This memory is configured such that both tasks have access to the shared section, each of them as part of their specified execution context. This is followed by initiating a notification, which initiates immediate transfer of control to the other world.

Notifications, also called signals or events, are required to have a corresponding synchronization mechanism. Signals, like interrupts, occur asynchronously and need

to inform a thread to process the results of the event. On a TrustZone platform, these notifications are realized through hardware interrupts.

The figure below shows an example of communication between the Normal-World and Secure-World. The Trusted Application is always rather passive, being the slave, whereas the TLC is responsible for opening a session to a Trusted Application and triggering it with notifications, acting as master.

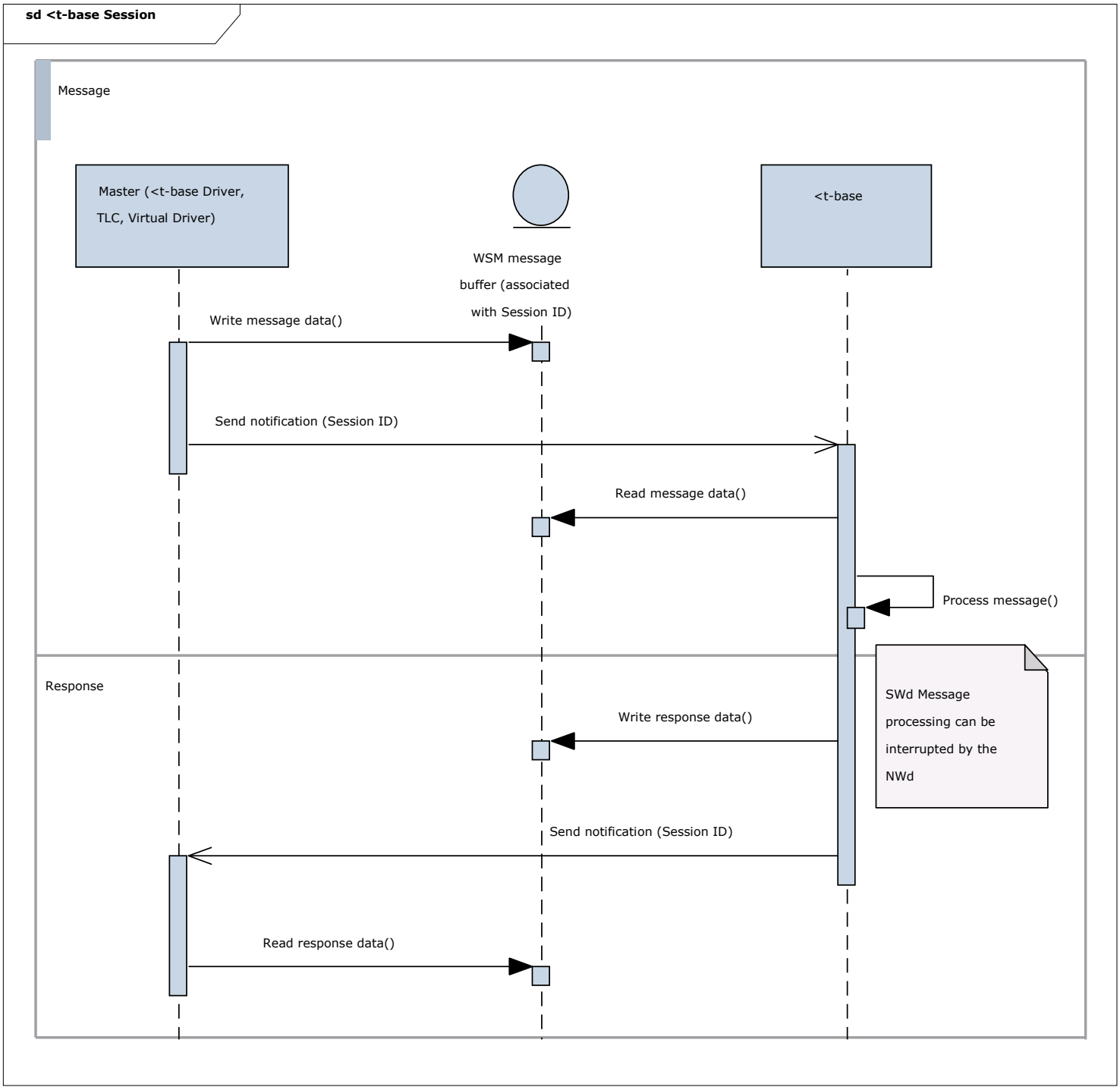


Figure 4: Inter-World Communication.

3.5.3 <t-base Communication Interface

The following sections provide a brief description of the <t-base Client API [API] functions to be used by the TLC for communication with the Trusted Application. See the API description [API] for the full function definitions.

3.5.3.1 Device Sessions

Before communicating with a Trusted Application, the TLC needs to open a session to a <t-base device. Currently there is only one standard device available (deviceId = MC_DEVICE_ID_DEFAULT), but in the future this may be extended by other TrustZone runtime environments, or Secure Elements providing the same API.

```
mcResult_t mcOpenDevice(
[in] uint32_t deviceId
);

mcResult_t mcCloseDevice(
[in] uint32_t deviceId
);
```

The counterpart to the `mcOpenDevice()` command is `mcCloseDevice()`, freeing all still allocated device resources. Trusted Application sessions need to be closed prior to closing the device.

3.5.3.2 Trusted Application Sessions

A Trusted Application session is the initial communication channel to the Trusted Application. It is required for any subsequent signaling and message passing actions.

```
mcResult_t mcOpenTrusted Application(
    mcSessionHandle_t  *session,
    mcSpid_t           spid,
    uint8_t            *Trusted Application,
    uint32_t           tLen,
    uint8_t            *tci,
    uint32_t           tciLen
);
```

`mcOpenTrusted Application()` opens a session to a Trusted Application specified by the Service Provider ID and returns a handle for the session. It is required that a device has already been opened before. You have to prepare the 'session' structure's deviceId field with the deviceId that you opened before. The caller must provide the Trusted Application binary so that the Trusted Application is loaded in <t-base. The caller must also allocate the communication buffer before calling this function.

```
mcResult_t mcCloseSession(
[in] mcSessionHandle_t *session
);
```

`mcCloseSession()` closes a session and all its allocated resources.

3.5.3.3 Message Exchange and Signaling

<t-base's Inter World Communication signaling part is based on two functions:

```
mcResult_t mcNotify(
```

```
[in] mcSessionHandle_t *session
);
```

`mcNotify()` sends a signal to a Trusted Application to inform it about the availability of new data within the WSM buffer.

```
mcResult_t mcWaitNotification(
[in] mcSessionHandle_t *session,
[in] int32_t timeout
);
```

The TLC uses `mcWaitNotification()` to wait for the Trusted Application to respond. The timeout parameter sets the maximum amount of time that the TLC will wait for a certain Trusted Application session to respond.

Errors in the Secure World can be queried with the `mcGetSessionErrorCode()` call. See [API] for further details.

3.5.3.4 Memory Mapping

Memory mapping enables provisioning large amounts of data to a Trusted Application with zero copy.

```
mcResult_t mcMap(
[in] mcSessionHandle_t *sessionHandle,
[in] void *buf,
[in] uint32_t len,
[out] mcBulkMap_t *MapInfo
);
```

`mcMap()` maps additional memory to a Trusted Application. A TLC should use this whenever it needs to provide a Trusted Application with data stored in Normal-World memory (like a buffer holding the clear text for a cryptographic operation). Altogether 4 chunks of max. 1MB each can be mapped between a TLC and a Trusted Application. See also Fig. 2 Trusted Application Address Space Layout.

Hint: The amount of memory blocks concurrently mapped to the Trusted Application is limited to four (4). Use the mcUnmap(…) function to unmap not in use memory blocks.

```
mcResult_t mcUnmap(
[in] mcSessionHandle_t *sessionHandle,
[in] void *buf,
[in] mcBulkMap_t *MapInfo
);
```

`mcUnmap()` is the counterpart to the mapping function and unmaps previously mapped blocks of memory.

4 USING THE <T-SDK

4.1 TOOLCHAIN INSTALLATION

4.1.1 System Requirements

For the development PC, TRUSTONIC recommends at least a 2.0GHz CPU and 1-2GB RAM.

4.1.2 Trusted Application development

Trusted Application development requires a Shell environment.

We recommend using Ubuntu 12.04 or later (www.ubuntu.com).

It is also possible to use MinGW + MSYS (http://www.mingw.org/) under Microsoft Windows 7.

4.1.2.1 GCC / Linaro Compiler

The Linaro GCC 4.7.3 is GNU development toolchain for ARM Embedded Processors. It includes compiler tools and debugger for ARM base platforms. It is the preferred toolchain.

- For **Ubuntu**, download this archive:

gcc-arm-none-eabi-4_7-2013q1-20130313-linux.tar.bz2

- Copy the archive to t-sdk-rXXX/Tools/ folder
- Decompress the archive

```
tar xjf gcc-arm-none-eabi-4_7-2013q1-20130313-linux.tar.bz2
```

1. For Windows, download this archive:

gcc-arm-none-eabi-4_7-2013q1-20130313-win32.zip

2. Copy the archive to t-sdk-rXXX/Tools/ folder
3. Decompress the archive

```
unzip gcc-arm-none-eabi-4_7-2013q1-20130313-win32.zip
```

4.1.2.2 DS-5™ / ARM Compiler

The ARM Development Studio 5 (DS-5™) is a development environment including compiler and debugger for ARM base platforms. DS-5™ can be obtained from ARM. Please refer to the ARM web site http://www.arm.com (Products/Tools/Software Tools) for further details.

Installing the whole DS-5™ package is not necessary, as only the ARM Compiler is required for building Trusted Applications.

- install ARM-DS5 compiler toolchain 5 from https://silver.arm.com/browse/DS500/
- obtain ARM-DS5 license from ARM or internal lic-server (30 days eval license needs your MAC-address)

For debugging Trusted Applications on development platforms via JTAG, the ARM RV Debugger can be used.

Hint: Notice that the installation target directory should not contain any spaces. Otherwise, DS-5™ may not work correctly.

4.1.3 Client Application development with Android SDK / NDK

- Download and install the Android SDK suitable for your development platform from http://developer.android.com/sdk/index.html
  - Add the Android SDK platform as described in http://developer.android.com/sdk/installing/adding-packages.html
  - For Windows add the USB driver as described in http://developer.android.com/sdk/win-usb.html.
  - In case you need to connect to a non-standard development board, the version provided with the platform must be used, so that ADB is able to recognize the ID of the device.
- Download the Android NDK (version r8e or above) suitable for your development platform from http://developer.android.com/tools/sdk/ndk/index.html and install it (means uncompressing into a directory on your computer). The **NDK_BUILD** environment variable must point to android-ndk-rXX/ndk-build

Hint: if you are using Ubuntu 64-bit OS, make sure that the ia32-libs are installed. See http://packages.ubuntu.com/de/lucid/ia32-libs

Otherwise building the TLC together with NDK may not work.

4.1.3.1 Using ADB

The Android Debug Bridge (ADB) is required to connect a host PC to the Android device. It is part of the Android SDK.
On Linux set your path variable to point to the folder with the ADB binary:

```
PATH="$PATH:<path-to-SDK>/platform-tools"
```

4.1.4 Client Application development with Microsoft Visual Studio for Windows Desktop

Download and install Visual Studio for Windows Desktop.

http://www.microsoft.com/visualstudio/eng/downloads#d-2012-express

The <t-sdk package includes the header files and libraries necessary to compile and link Windows Client Applications with the <t-base client API.

- The main header file is "t-sdk-rXXX\t-sdk\TlcSdk\Public\MobiCoreDriverApi.h" and is not specific to the Windows platform.
- For **32**-bit Client Applications, the <t-base client API library is located here:
  - "t-sdk-rXXX\t-sdk\TlcSdk\Bin\Windows**Win32**\Release\t-base_client_api.lib"
- For **64**-bit Client Applications, the <t-base client API library is located here:
  - "t-sdk-rXXX\t-sdk\TlcSdk\Bin\Windows**x64**\Release\t-base_client_api.lib"

It is strongly recommended to have a look at the sample projects.

For example, run:

```
t-sdk-rXXX\Samples\Aes\TlcAes\Locals\Code\Windows\TlcSampleAes.bat
```

And compile the solution.

It exists another script allowing you to build all samples in a raw:

```
t-sdk-rXXX\Samples\build_client_applications_for_windows.bat
```

WARNING: these scripts have been generated to support Visual Studio version 11.0 only. Reworking some .vcxproj files might be required if you want to use a different version.

4.2 QUICK START GUIDE

This chapter contains a 'quick start guide' to <t-sdk.

Please read also the chapters:

- Trusted Application Development
- Trusted Application Connector Development

4.2.1 File structure

The <t-sdk package provides everything you need for TA and TLC development under Linux:

- Documentation\
  Documentation including API references and this guide.
- Samples\
  Sample projects to get started quickly.
- t-sdk\
  Contains header files and libraries to compile TLs and TLCs.
- \
  - index.html
    Documentation entry point to get started using the <t-sdk.
  - setup.sh
    Set the paths to your toolchains there.

4.2.2 Setup instructions

Below the setup/installation guide of <t-sdk to create sample Trusted Applications, TLCs and apps

- update paths in

  ```
  ~/workspace/t-sdk-rXXX/setup.sh
  ```

  to point to your toolchains and execute (needs execution rights) `setup.sh` in the top package folder lists the paths to the different needed components for the different Makefiles included in the release.

1. Set `COMP_PATH_AndroidNdk` to your Android NDK installation (r8 or later).
2. Set `ARM_RVCT_PATH` to your ARM DS-5 installation.
3. Set `LM_LICENSE_FILE` to your ARM license.
   The license can be related to by using:
   - the absolute path of the license file (example: `/home/user/ArmLicense/RVS41.dat`).
   - a license on a server (example: 42@42).
4. Check if your ARM DS-5 installation has the required subfolders "inc", "lib" and "bin/linux_x86_x32" and adopt the paths of `ARM_RVCT_PATH_*` if required.
5. Set `CROSS_GCC_PATH` to your extracted Linaro GCC directory

Example: `/opt/gcc-arm-none-eabi-4_7-2012q4`

- run setup.sh

```
./setup.sh
```

4.2.3 Creating the first secure application

Now you can use the SHA256 <t-sdk -Sample to create your own Trusted Application, TLC and Android app.

4.2.3.1 Trusted Application (TA)

- update the demo makefile.mk and build.sh for your own TA
- for printing debug output in your Trusted Applications, use functions in tlApiLogging.h (tlApiLogvPrintf tlApiLogPrintf tlDbgPrintf tlDbgvPrintf).
- call the build script

```
./build.sh
```

When a Trusted Application is built using this command, it will use the default options and will compile a Trusted Application in Debug with the ARM compiler.

- it is possible to specify which MODE and TOOLCHAIN will be used to build the Trusted Application

```
MODE      could be  Debug or Release
TOOLCHAIN could be  ARM   or GNU
```

The script should be called this way:

```
MODE=Debug TOOLCHAIN=ARM ./build.sh      #default values
MODE=Release TOOLCHAIN=GNU ./build.sh
```

4.2.3.2 Trusted Application Connector (TLC)

- update the demo Android.mk and build.sh for your own TLC
- to get the TLC build running, you have to export additional the path to the output folder of your TA

```
export COMP_PATH_<yourTLname>=
${COMP_PATH_ROOT}/../projects/<yourProject>/<youtTLdir>/O
ut
```

- call the build script

```
./build.sh
```

4.2.3.3 Android SDK

- open the Android SDK eclipse and import the android-app `project.properties`
- build the app

4.2.4 Running the first secure application

If you have a <t-base enabled device already, you can try to run your sample TLC and TA.

4.2.4.1 On a Commercial Device (running a SP-TA)

System Trusted Applications cannot be loaded on commercial devices for testing; the only way of running your Trusted Application is loading it as a SP-TA.

On commercial smartphones and tablets you need connection to the <t-base backend using the Provisioning Agent (PA).

Hint: working with commercial devices usually implies having root access rights i.e. to push files to /data/app/mcRegistry.

Use a SP Test Trusted Application container:

There is a TA in the Backend that can be used for testing your Trusted Application.

1. Use the XML key file that comes with the <t-sdk (`t-sdk-rXXX\Samples\Sha256\TlSampleSha256\Locals\Build\keySpTl.xml`) it contains the AES key of the 04010200..00 sample SP-TA container in the Backend:

```
<?xml version="1.0"?>
<Key>000102030405060708090a0b0c0d0e0f000102030405060708090a0b0c0d0e
0f</Key>
```

2. Compile your Trusted Application as a SP-TA with following options (`makefile.mk`)

```
TRUSTLET_UUID := 04010200000000000000000000000000
TRUSTLET_SERVICE_TYPE := 2 # SP Trusted Application
TRUSTLET_KEYFILE := <your path>/keySpTl.xml
```

3. Install the Provisioning Agent Android app.

4. A SP-container for your TA ("<UUID>.spcont") is needed on your device in `data/app/mcRegistry/` to run it as SP-TA
   ➔ Trigger the backend to create it:

```
adb shell ls /data/app/mcRegistry
adb shell am start -a "com.secunet.android.t-base.install_sp_tl" -e
"TL_UUID" "04010200-0000-0000-0000-000000000000"
```

5. load both modules (from TA/TLC out-folders) onto your device
   If mcOpenSession() is used by the TLC, the Trusted Application must be pushed to /data/app/mcRegistry otherwise when mcOpenTrusted Application() is used by the TLC, the Trusted Application must be pushed to the location expected by the TLC.

```
adb push 040102000000000000000000000000000.tlbin
data/app/mcRegistry/ #mcOpenSession() case
adb push <myTLCname> data/app/
```

6. start your TLC

```
adb shell data/app/<myTLCname>
```

4.2.4.2 On a Development Board (running as System-TA or SP-TA)

Development boards (Arndale are typically not known to the Backend but 16 SP-TA containers are predefined.

To execute your Trusted Application on a Development Board it has to be compiled either as System Trusted Application and signed by the dummy OEMs RSA private key or as Service Provider Trusted Application and signed with the development key provided in the <t-sdk.

Hint: <t-base images are available for some development platforms / boards. See [DevDev] for details or contact support@trustonic.com .

Use a signed System Trusted Application:

1. Use the RSA System TA sample private key that comes with the <t-sdk
   `(t-sdk-rXXX\Samples\Sha256\TlSampleSha256\`
   `Locals\Build\pairVendorTltSig.pem)`

2. Compile your Trusted Application as a System TA (adapt `makefile.mk`)

```
TRUSTLET_UUID := 060100000000000000000000000000000
TRUSTLET_SERVICE_TYPE := 3 # System Trusted Application
TRUSTLET_KEYFILE := <your path>/pairVendorTltSig.pem
```

3. load both modules (from TA/TLC out-folders) onto your device

```
adb push 060100000000000000000000000000000.tlbin data/app/mcRegistry/
adb push <myTLCname> data/app/
```

4. start your TLC

```
adb shell data/app/<myTLCname>
```

Use a signed SP Trusted Application: see previous chapter.

5 TRUSTED APPLICATION DEVELOPMENT

The following section describes what you need to get started writing your first Trusted Application, compiling it and loading it to the device.

<t-base's main feature, for you as a developer, is that it can load additional code during run-time. In detail, it allows writing small programs which get pushed to the device and started as a <t-base process when they are needed.

Developing a Trusted Application is similar to developing a C process, except for the fact that the Trusted Application is running in a separate environment. As described in Chapter 3.5, the communication between Trusted Applications and TLCs is done via notifications and shared memory. See the next section for a basic example of these IPC mechanisms.

5.1 BUILD ENVIRONMENT

For creating new Trusted Applications you can start with one of the existing Samples in the <t-sdk, which has the following layout:

`- Locals`	Holds everything that belongs to the Trusted Application.
`-- Build`	Holds the Trusted Application build script. A build is started via `build.sh`. It sets the build environment and starts the build process.
`-- Code`	Holds the Trusted Application code. Public header files should be placed in a subfolder called `public`. The compilation parameters can be set in `makefile.mk`.
`- Out`	Holds the generated Trusted Application binary.
`-- Bin`	Generated binaries
`-- Public`	Exported headers

5.2 WRITE THE TRUSTED APPLICATION CODE

5.2.1 Trusted Application Structure

A Trusted Application implements a fairly simple startup, main loop, and shutdown process. See the example code provided with the <t-sdk.

An example of a basic Trusted Application main routine block would be something like:

```
void tlMain(
uint8_t *tciData,
uint32_t tciLen
) {
      // Check TCI size
      if (sizeof(tci_t) > tciLen) {
            // TCI too small -> end Trusted Application
            tlApiExit(EXIT_ERROR);
      }
      // Trusted Application main loop
      for (;;) {
            // Wait for a notification to arrive
            tlApiWaitNotification(INFINITE_TIMEOUT);

            // ************** //
            // Process command
            // ************** //

            // Notify the TLC
            tlApiNotify();
      }
}
```

The `tlMain` routine has two parameters:

- The first one is a pointer to the world shared memory TCI buffer, allocated by the <t-base driver during the initialization of a new session.
- The later provides the length of the buffer. This must be something below 4kB, which is the maximum length of an allocateable block of memory.

After a successful initialization (where you should check if the provided TCI memory is big enough) the Trusted Application will loop infinitely untill its process is killed by the <t-base runtime due to a close command from the TLC or a fatal exception.

In its main loop, the Trusted Application will repeatedly:

1. wait for a notification from the TLC via the `tlApiWaitNofitification()` command

Hint: Infinite timeout is recommended, not polling.

2. process the data from the TCI buffer and write back the result to the buffer
3. signal the TLC that there is a response available with `tlApiNotify()`

This pretty much looks the same for every Trusted Application. Have a look at the samples at the end of this document for more details about this approach.

5.2.2 Advanced TCI Communication Protocol

If your Trusted Application needs to handle more than one command, or needs to support a more comprehensive communication protocol, it is advised to add a command / response protocol on top of the TCI. This kind of protocol allows the Trusted Application interface to define commands and responses reflecting the provided functionality of the Trusted Application. The messages need to be exchanged via the TCI world share memory buffer, established by the TLC during the creation of a new Trusted Application session.

See for a suggestion of such a protocol.

Pos	Type	Size	Description
1	Header	uint32_t (4 bytes)	Header may be: 0 < Command ID < 0x7fffffff, or a Response ID (Command ID | 0x80000000)
2	Payload	TCI_SIZE – 4	Holds the command or response data

Tab. 1 TCI command- and response protocol structure

The first field in the communication structure would hold the header which would either be a command for the Trusted Application or a response for the TLC.

The second field defines a payload field to hold the data belonging to the corresponding header.

Hint: The length of the data can be of arbitrary size, but the header and data field must fit in the maximum TCI buffer size, which is currently at 4kB.

For each function your Trusted Application provides to the TLC, you would define a command ID, along with a specific data structure for the payload.

Here is an excerpt of a Trusted Application header file (API) of the definitions of command IDs (TCI definition) as well as their data structures:

```
/** Command ID's */
#define CMD_SAMPLE_SHA1          1
#define CMD_SAMPLE_SHA256   2

/** Message digest command (SHA1 or SHA256) */
typedef struct {
    tciCommandHeader_t commandId;
    uint8_t* srcBuffer;
      uint32_t srcLen;
} cmdMD_t;
```

The code shows the API for a Trusted Application providing the SHA1 and SHA256 hash functions to the Normal-World. Both hashes share the same command structure:

- An ID to tell the Trusted Application which kind of hash we want

- A pointer to the source buffer holding the data we want to calculate a hash from
- The length of the input data

As the two hash functions differ in their resulting output size, two separate response structures have been defined, both starting with the header field followed by the hash result.

```
/** SHA1 response */
typedef struct {
    tciResponseHeader_t responseID;
    uint8_t hash[20];
} rspSha1_t;

/** SHA256 response */
typedef struct {
    tciResponseHeader_t responseID;
    uint8_t hash[32];
} rspSha256_t;
```

Now that all possible command and response structures have been defined, declare the TCI message as a Union over them:

```
/** TCI message data. */
typedef union {
    tciCommandHeader_t      commandHeader;
    tciResponseHeader_t     responseHeader;
    cmdMD_t                 cmdMD;
    rspSha1_t               rspSha1;
    rspSha256_t            rspSha256;
} tciMessage_t;
```

The Union allows the Trusted Application to easily access the command ID within the TCI buffer and map it to the according command structure.

On top of that, the Trusted Application can define various error codes:

```
#define RET_ERR_INVALID_BUFFER        3
#define RET_ERR_INVALID_COMMAND  4
```

Hint: Notice that there is no need for an "OK" return code for the good case. Using a bit mask flipping the most significant bit of the command ID is the suggested way to signal the TLC a valid response to a certain command.

Now getting back to the previously described `tlMain` loop, applying the new definitions would change it in the following way:

1. Whenever the Trusted Application receives a notification, it first checks the provided command ID from the TCI. If the command ID is unknown, the Trusted Application sets the corresponding response ID and the payload to RET_ERR_INVALID _COMMAND.
2. The Trusted Application executes the corresponding command.

3. If an error occurs during the execution of a command the header field will be set to an error value and an empty response will be returned.
4. In case the Trusted Application function did return successfully, the command ID is converted to a response ID and written to the response header. Furthermore, the response data structure will be returned in the payload field.

Here is the resulting code:

```
/**< Responses have bit 31 set */
#define RSP_ID_MASK (1U << 31)
#define RSP_ID(cmdId) (((uint32_t)(cmdId)) | RSP_ID_MASK)
#define IS_CMD(cmdId) ((((uint32_t)(cmdId)) & RSP_ID_MASK) == 0)
#define   IS_RSP(cmdId)   ((((uint32_t)(cmdId))   &   RSP_ID_MASK)   ==
RSP_ID_MASK)

void tlMain(
uint8_t *tciData,
uint32_t tciLen
) {
      tciResponseHeader_t responseId;
      tciCommandHeader_t commandId;

      // Check TCI size
      if (sizeof(tci_t) > tciLen) {
            // TCI too small -> end Trusted Application
            tlApiExit(EXIT_ERROR);
      }

      tciMessage_t* tciMessage = (tci_t*) tciData;
      // Trusted Application main loop
      for (;;) {
            // Wait for a notification to arrive
            tlApiWaitNotification(INFINITE_TIMEOUT);

            commandId = tciMessage->commandHeader.commandId;

            // Check if the message received is (still) a response
            if (!IS_CMD(commandId)) {
                  // Tell the Normal-World a response is still pending
                  tlApiNotify();
                  continue;
            }
            // Call Trusted Application functions according to command ID
            switch (commandId) {
            case CMD_SAMPLE_SHA1:
                  ret = processCmdSha1(tciMessage->cmdMD);
                  break;
            case CMD_SAMPLE_SHA256:
                  ret = processCmdSha256(tciMessage->cmdMD);
                  break;

            default:
                  ret = RET_ERR_UNKNOWN_COMMAND;
```

```
                break;
            }

            // Set up response header
            tciMessage->responseHeader.responseId = RSP_ID(commandId);
            tciMessage->responseHeader.returnCode = ret;

            // Notify back the TLC
            tlApiNotify();
        }
}
```

5.2.3 Security Considerations

The Trusted Application's code and data regions lie within the first 1 MB of its memory / address space. Any constant data structures linked into the Trusted Application binary will be mapped to this memory area. Depending on the type of Trusted Application this may include security critical data like keys, cryptographic seeds, PINs…

Note that this 1MB will change in the future versions of <t-base and it is recommended to call `tlApiGetVirtMemType`() to check if a buffer is only accessible by the Secure-World or shared with the Normal-World.

<t-base and the TrustZone concept will prevent direct access to Secure-World memory by the Normal-World, but it cannot stop a TA accidentally leaking security relevant data by copying it to WSM regions.

Using the SHA256 sample Trusted Application provided in this document, one could think of the following attack scenario:

- The Trusted Application calculates a hash over a memory area, it gets mapped by the TLC and is informed by the `srcBuffer` pointer element of the `cmdMD_t` structure.
- The Trusted Application assumes that this buffer has been mapped by the TLC using the <t-base Driver API prior to notifying the Trusted Application. From this it follows, that the buffer would lie between 1MB and 8MB of the Trusted Applications memory space.
- As the Trusted Application never verifies that this is actually the case, a malicious TLC could point the Trusted Application to its own code or data region, providing a pointer with a value below 1MB (< 0x100000)!
- Although the Trusted Application only returns the SHA256 hash of this memory, a TLC could repeatedly call this function only hashing one byte at a time.
- Comparing each result with a table containing the hash results for the values 0 – 256 (a hash map), one can quickly derive the content of the memory address.
- This would allow an attacker to read the clear text Trusted Application binary, including code and data, like keys!
- Trusted Applications cannot use NEON registers. <t-base does not enable the NEON extension and respective instruction will cause an undefined instruction abortion and terminate the Trusted Application.

Note: It is vital to carefully define and implement the TA API to protect direct or indirect access to internal memory structures. Trusted Application should use tlApiGetVirtMemType to test memory type, and accept only pointers pointing to Normal-World memory.

5.2.4 <t-base Internal API

The <t-base Internal API is used by Trusted Applications. It defines all the functionality the <t-base can offer to a Trusted Application:

- A set of functions for inter-world communication.
- <t-base system information and functions.
- Cryptographic processing.
- Secure object functions for binary data encryption.

Please see [API] for complete reference.

5.2.5 Secure Objects

For storing Trusted Application data persistently, <t-base provides the Secure Object API functions. Secure objects are wrapped and unwrapped using internal <t-base keys and therefore provide convenience functions for secure data storage in the Normal-World.

Major Secure Object functionalities are:

- **Data Integrity.** A Secure Object contains a message digest (hash) that ensures data integrity of the user data. The hash value is computed and stored during the wrap operation (before data encryption takes place) and recomputed and compared during the unwrap operation (after the data has been decrypted).
- **Confidentiality.** Secure Objects are encrypted using context-specific keys that are never exposed, neither to the normal world, nor to the Trusted Application. It is up to the user to define how many bytes of the user data are to be kept in plain text and how many bytes are to be encrypted. The plain text part of a Secure Object always precedes the encrypted part.
- **Authenticity.** As a means of ensuring the trusted origin of Secure Objects, the unwrap operation stores the Trusted Application ID (SPID, UUID) of the calling Trusted Application in the Secure Object header (as Producer). This allows Trusted Applications to only accept Secure Objects from certain partners. This is most important for scenarios involving secure object sharing.

Design Considerations:

- The TLC is responsible for storing the Secure Object in Normal-World. <t-base does not support direct access to the rich OS file system. Therefore the TA needs to wrap the Secure Object into WSM and notify the TLC to store it persistently. Later the TLC needs to provide Secure Object to the TA via WSM once needed.
- Secure Objects can be bound to the TA itself or a broader scope. This enables devices binding scenarios or as well as sharing data securely between TLs (e.g. via a common TLC).
- Having an unencrypted, but signed, data part enables the implementation of a header or meta-data part. This can simplify Secure Object handling in both worlds.

Note: Source of wrap must be in internal TA memory (static buffers, BSS). Destination of unwrap must be in internal TA memory. Don't do in-place decryption in WSM to avoid leaking unencrypted data!

The API is composed of the following two operations:

```
tlApiResult_t tlApiWrapObject(
    const void *src,
    size_t plainLen,
    size_t encryptedLen,
    mcSoHeader_t *dest,
    size_t *destLen,
    mcSoContext_t context,
    mcSoLifeTime_t lifetime,
    const tlApiSpTrusted ApplicationId_t *consumer);

tlApiResult_t tlApiUnwrapObject(
    mcSoHeader_t *src,
    void *dest,
    size_t *destLen);
```

Secure Object Context

The concept of context allows for sharing of Secure Objects. There are three kinds of context:

- MC_SO_CONTEXT_TLT: Trusted Application context. The secure object is confined to a particular Trusted Application. This is the standard use case.
  - PRIVATE WRAPPING: If no consumer was specified, only the Trusted Application that wrapped the Secure Object can unwrap it.
  - DELEGATED WRAPPING: If a consumer Trusted Application is specified, only the Trusted Application specified as 'consumer' during the wrap operation can unwrap the Secure Object. Note that there is no delegated wrapping with any other contexts.
- MC_SO_CONTEXT_SP: Service provider context. Only Trusted Applications that belong to the same Service Provider can unwrap a secure object that was wrapped in the context of a certain service provider.
- MC_SO_CONTEXT_DEVICE: Device context. All Trusted Applications can unwrap secure objects wrapped for this context.

Secure Object Lifetime

The concept of a lifetime allows limiting how long a Secure Object is valid. After the end of the lifetime, it is impossible to unwrap the object.

Three lifetime classes are defined:

- MC_SO_LIFETIME_PERMANENT: Secure Object does not expire. HINT: Only if the platform supports the required hardware features (like Secure-World non-volatile monotonic counters), this is protected against replay attacks.
- MC_SO_LIFETIME_POWERCYCLE: Secure Object expires on reboot.

- MC_SO_LIFETIME_SESSION: Secure Object expires when Trusted Application session is closed. The secure object is thus confined to a particular session of a particular Trusted Application. Note that session lifetime is only allowed for private wrapping in the Trusted Application context MC_SO_CONTEXT_TLT.

Secure Object Consumer

The consumer parameter is only valid for the context MC_SO_CONTEXT_TLT (DELEGATED WRAPPING). It defines which Trusted Application (SPID, UUID) is allowed to unwrap that Secure Object.

This identity (of type tlApiSpTrustedApplicationId_t) is used for delegation purposes and to which Trusted Application we delegate the secure object, i.e. which Trusted Application should unwrap the secure object. Here we state that the Trusted Application (and only that Trusted Application) having the UUID `"0,9,0,9,0,0,0,0,0,0,0,0,0,0,0,0"` can unwrap the secure object which will be saved in the normal world.

```
static const tlApiSpTrustedApplicationId_t consumerTid = {
    0,
    { 9,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
};

ret = tlApiWrapObject(userdata,
                      UDATA_PLAIN_LEN,
                      UDATA_ENC_LEN,
                      (mcSoHeader_t *)soDataBuf,
                      &soLength,
                      MC_SO_CONTEXT_TLT,
                      MC_SO_LIFETIME_PERMANENT,
                      &consumerTid,
                      TLAPI_WRAP_DEFAULT);
```

The TLAPI_WRAP_DEFAULT flag is mapped (using `#define`) in the <t-base and the Trusted Application developer should use this flag when calling the wrapper function.

When wanting to unwrap the secure object, this is done within the Trusted Application of the UUID ”`0,9,0,9,0,0,0,0,0,0,0,0,0,0,0,0`”.

The Trusted Application calls:

```
ret = tlApiUnwrapObject((mcSoHeader_t*)pMessage->data,
                        pMessage->cmdSOUnwrapDelegate.len,
                        userdataDest,
                        &userdataDestLength,
                        TLAPI_UNWRAP_PERMIT_DELEGATED|
                        TLAPI_UNWRAP_DEFAULT);
```

The description of the use case is described below.

Precondition: The Trusted Application container is installed.

Starting point: Trusted Application Connector A is in waiting for data from Trusted Application A. Trusted Application B is waiting for data from Trusted Application Connector B.

1. Data buffer and identity (for delegation) of Trusted Application B is sent as input to tlApiWrapObjectExt
2. The data buffer is wrapped and (some part) encrypted
3. The wrapped data (SO) is sent to TLC A
4. The TLC A saves the data to file
5. The SO is read from file by TLC B
6. The SO is sent to Trusted Application B
7. Trusted Application B calls tlApiUnwrapObjectExt using the SO as input.
8. The data buffer (see 1.) is returned. The Security functionality checks if Trusted Application B has the right to decrypt the SO. Since the identity of Trusted Application B was set (see 1.) this is possible.

5.2.6 Checking API return value

Most of the API calls return error code of type `tlApiResult_t`.

It always recommended to check that returned value is TLAPI_OK.

In some cases, Trusted Application may want to check for a specific error.

This checking should be implemented using the `TLAPI_ERROR_MAJOR` macro.

The macro returns the stable part of the error code. The remaining part of error code contains detail code, which may change between <t-base releases.

5.3 COMPILING AND SIGNING A TRUSTED APPLICATION

Trusted Applications are directly run on the target platform without any interpretation. Thus, they can be written in assembly code, C or in any other high level language where a compiler exists.

Using the provided <t-sdk, Trusted Applications need to be written in C according to the C99 standard.

The ARM Compiler is the recommended tool chain for compiling and linking. Furthermore, the ARM Library is used for basic header files and functions.

GNU GCC toolchain is also supported by the <t-sdk. [0]

Other tool chains can be used to build Trusted Applications too, but currently there is no support for them in the <t-sdk.

Prerequisites:

- Select a new UUID for your Trusted Application. On a development environment, the UUID can be selected freely according to RFC 4122 from the

<t-sdk Development Trusted Application UUIDs listed in the table below. Please use MobiConvert to generate a proper UUID, see section Mobiconvert Manual.

- Generate an AES256 key (32 byte) and enter it in the <Key> tag in key.xml in ASCII hexadecimal notation.
- See the section Mobiconvert Manual for more information about their format.

For use of <t-sdk development targets following UUIDs are predefined and must be used:

Name	UUID	Description
Sample ROT13	0401-0000-0000-0000-0000-0000-0000-0000	**sample code**
Sample SHA256	0601-0000-0000-0000-0000-0000-0000-0000	**sample code**
TlAes	0702-0000-0000-0000-0000-0000-0000-0000	**sample code**
TlRsa	0704-0000-0000-0000-0000-0000-0000-0000	**sample code**
<t-sdk Development	0801-0000-0000-0000-0000-0000-0000-0000 0802-0000-0000-0000-0000-0000-0000-0000 0803-0000-0000-0000-0000-0000-0000-0000 0804-0000-0000-0000-0000-0000-0000-0000 0805-0000-0000-0000-0000-0000-0000-0000 0806-0000-0000-0000-0000-0000-0000-0000 0807-0000-0000-0000-0000-0000-0000-0000 0808-0000-0000-0000-0000-0000-0000-0000 0809-0000-0000-0000-0000-0000-0000-0000 080A-0000-0000-0000-0000-0000-0000-0000 080B-0000-0000-0000-0000-0000-0000-0000 080C-0000-0000-0000-0000-0000-0000-0000 080D-0000-0000-0000-0000-0000-0000-0000 080E-0000-0000-0000-0000-0000-0000-0000 080F-0000-0000-0000-0000-0000-0000-0000 0810-0000-0000-0000-0000-0000-0000-0000	for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only reserved reserved reserved reserved

Tab. 2 <t-sdk UUIDs for Development targets

Developers must use for development target a UUID among the 12 first UUIDs used for testing from 0801-0000-0000-0000-0000-0000-0000-0000 until 0810-0000-0000-0000-0000-0000-0000-0000.

The 4 last UUIDs of the table are reserved for Trustonic and should not be used.

Hint: For production use you should use a key with sufficient entropy and a UUID which can be used throughout the life time of the TA.

Using the <t-sdk you compile and sign a Trusted Application using the Build process included in the sample Trusted Application structure, in two easy steps.

1. Write your parameters in makefile.mk, which can be found in the Code folder of the Trusted Application (for more information about the possible parameters for MobiConvert, see section Mobiconvert Manual).

2. Start the build process by running build.sh, which is in the build folder of the sample Trusted Application (in case you did not do so before, you should adapt setup.sh to your environment).

5.4 RUNNING A TRUSTED APPLICATION IN A DEVELOPMENT ENVIRONMENT

Whenever a TLC is requests a session to a Trusted Application specified by a UUID, the <t-base Driver looks for a Trusted Application with this UUID on the file system and loads it. Therefore, the following steps are necessary to run a Trusted Application.

1. Make sure that the output binary file of the Trusted Application build process is named after your selected UUID. For example, if the UUID is {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8} the file named becomes "01020304050607080102030405060708.tlbin".
2. Using ADB, push the Trusted Application to the `/data/app/mcRegistry` folder on the device, so that the Trusted Application can be found and loaded by the <t-base Driver.

   If `mcOpenTrusted Application`() is used, the Trusted Application does not have to be in `/data/app/mcRegistry`.
3. Develop a TLC that uses the Trusted Application (see section 7).

5.5 DEBUGGING A TRUSTED APPLICATION

<t-base is a secure runtime environment and thus there are limited debugging options on the target platforms.

In fact, debugging the Secure-World may not be allowed at all on end user platforms.

Hint: Actively debugging Trusted Applications bypasses security restrictions enforced by <t-base and TrustZone. As a consequence, a debug-able system can no longer be considered as a secure environment. Therefore it is strongly recommended that any involved background system is aware that it is dealing with a non-secure development platform.

5.5.1 Logging

Logging to a TCI or other WSM remains the only guaranteed option for monitoring a Trusted Application. State information can be read from TCI/WSM by a TLC and then printed or logged to a file.

A log-enabled version of <t-base may exist for target platform, which prints runtime information to a dedicated channel like UART, display or even JTAG-DCC. Trusted Applications can be linked against a log-enabled TlApi library which provides the function `tlDbgChar()` and `tlDbgBuffer()`. Furthermore, convenience functions like `tlDbgPrintf()` are available, but their usage can require an additional amount of stack.

5.5.2 Development Platforms

Special development versions of the target platform and generic development platforms exist. They allow full JTAG access, so that any code can be debugged in detail. However, halting the system or manipulating certain registers may change the runtime behavior of the system significantly.

This may even lead to an unstable system due to timing issues.

Furthermore, <t-base is a multi-tasking system which uses the MMU. Every task runs in a separate virtual address space. JTAG access usually does not bypass the MMU, thus breakpoints are set on virtual addresses only.

Usually, a debugger is not aware of different tasks and address spaces by default. Thus manual checks of the current task are necessary each time a breakpoint is hit.

Hint: Please refer to the manual of your debugger for details about process- or task-aware debugging.

5.5.3 Debug Agent

5.5.3.1 DS-5 setup

5.5.3.1.1 Install DS-5 Version 5.9.

- It is available at the ARM download page: DS-5 download, this requires an ARM account

5.5.3.1.2 Obtain a license

- DS-5 comes with 30 days evaluation license.

5.5.3.2 Install debug agent

- Launch DS-5
- Go to '`Window->Preferences->DS-5->Target Database`'. (See '`Window->Preferences->DS-5->Configuration Database`' on DS-5 v5.12 and onwards)
- Click '`Add...`', browse to t-sdk-rXXX\Tools\DebugExtension
- Click '`Apply`', then click "`Rebuild database`"
- Restart DS-5

5.5.3.2.1 Debug agent setup on PC host

- Go to '`Debug Control`' view by choosing, select right click and '`Debug Configurations...`'
  - The 'Debug Configurations' is found in the 'Run' menu.
- Under '`DS-5 Debugger`', create a new configuration and name. This configuration has to be set up so that you can connect to the board and also step in the code. You should now have six different tabs:
  - Connection
  - Files
  - Debugger
  - Arguments

- Environment
- Event Viewer

Connection tab

- Select "Generic - MobiCore_Android" as the Platform. Keep the default parameters, as most of them are currently ignored, and just manually add the IP address of your target. *Note:* If the Android device (e.g. the board) is connected via a USB cable please use **adb forward tcp:<port> tcp:<forward>** e.g. **adb forward tcp**:3010 tcp:3010. In this case you should use the IP address tcp:127.0.0.1 and port 3010.
- Under the '`Connections`' tab, you will see Trusted Application UUID. It is set to '`04010000000000000000000000000000`' as default. If you would like to change Trusted Application UUID, you need to manually modify '`Boards\Generic\MobiCore_Android\trusted_app.rvc`' and replace all occurrences of '`04010000000000000000000000000000`' to UUID of your Trusted Application OR just use the helper script `prepare_debug.sh`

Files tab

- In the 'Files' frame choose the 'File System' button.
  - Choose the axf file that lies in the Trusted Application project which you wish to run/debug, for example:

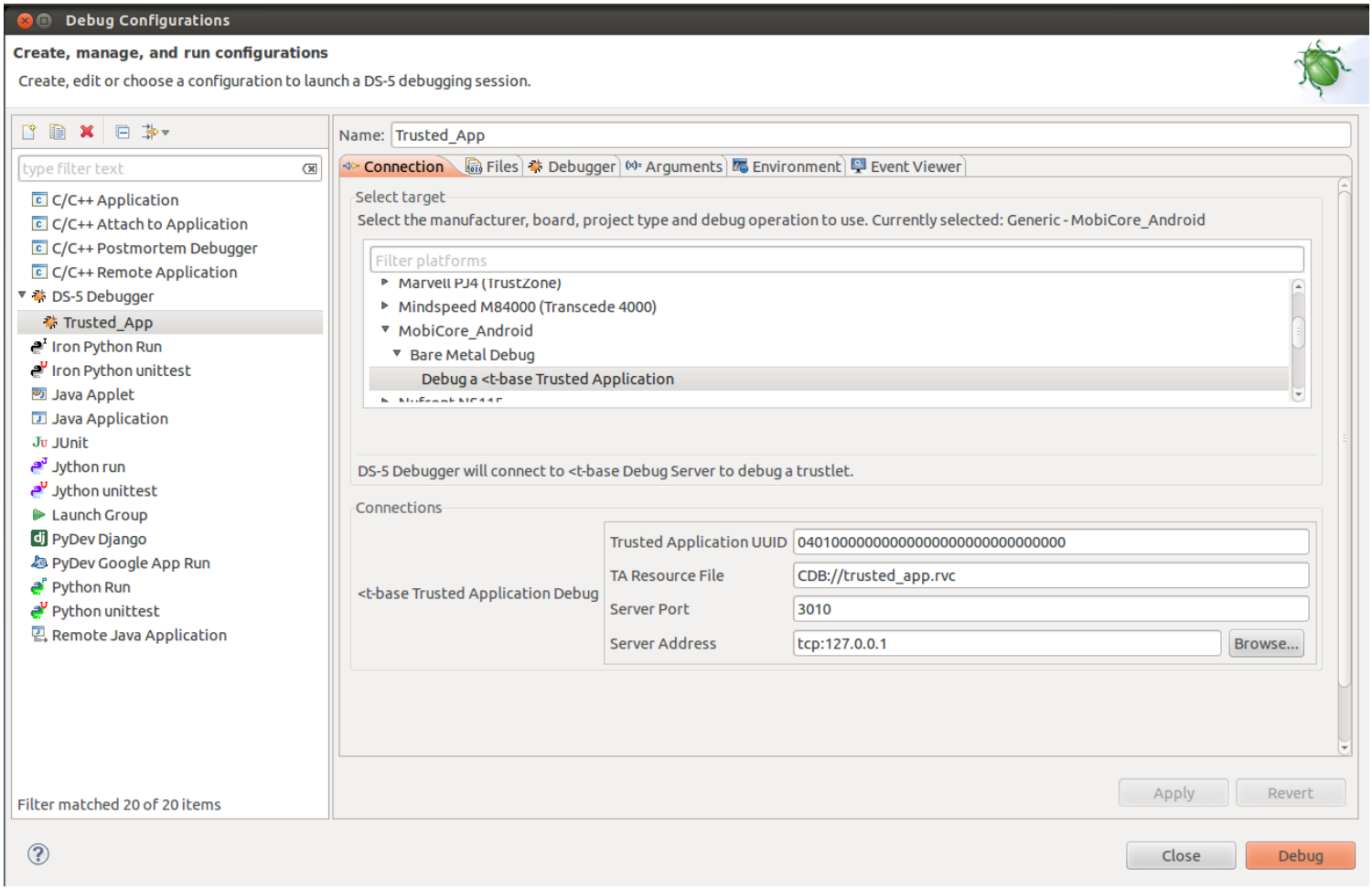
```
t-sdk-rXXX/Samples/Rot13/TlSampleRot13/Out/Bin/Debug/tlSampleRot13.axf
```

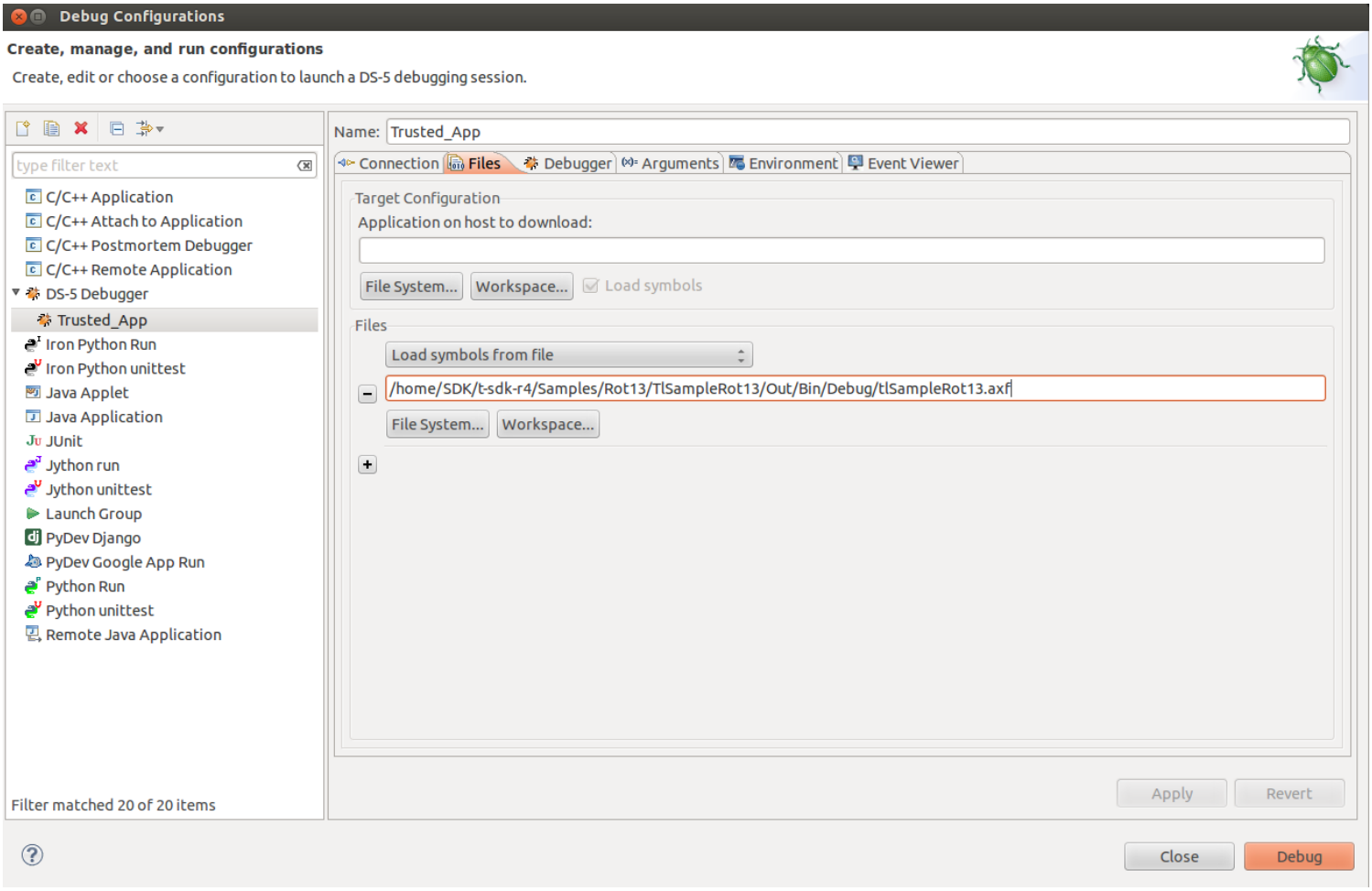
Debugger tab

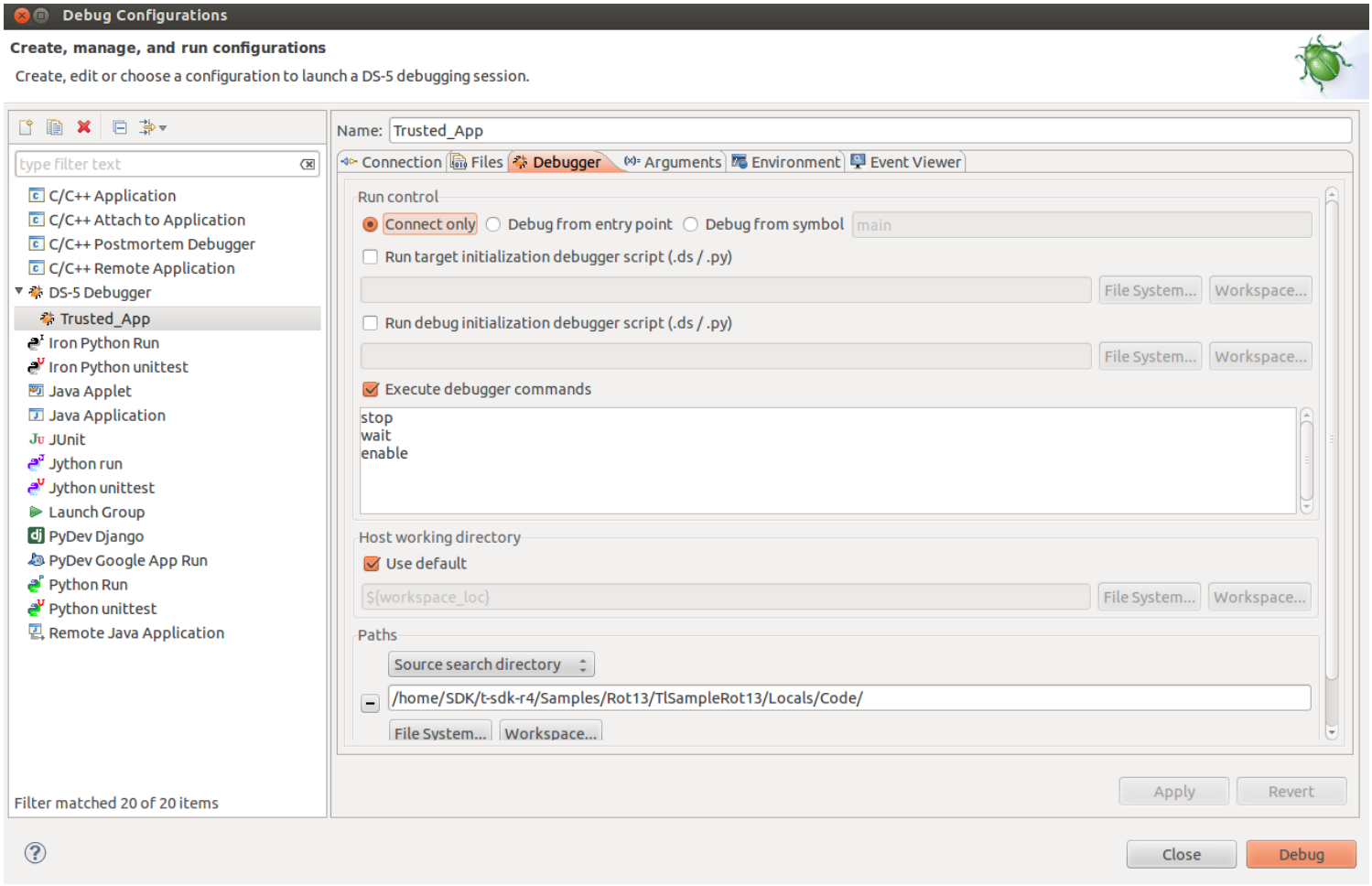
- Under the '`Run control`' , select '`Connect only`' for '`Run control`'.
- Check '`Execute debugger commands`' and the following:

```
stop
wait
enable
```

- Customize the directory of the source code folder.


Debug Configurations
Create, manage, and run configurations
Create, edit or choose a configuration to launch a DS-5 debugging session.
type filter text
C/C++ Application
C/C++ Attach to Application
C/C++ Postmortem Debugger
C/C++ Remote Application
DS-5 Debugger
Trusted_App
Iron Python Run
Iron Python unittest
Java Applet
Java Application
JUnit
Jython run
Jython unittest
Launch Group
PyDev Django
PyDev Google App Run
Python Run
Python unittest
Remote Java Application
Filter matched 20 of 20 items
Name: Trusted_App
Connection
Files
Debugger
Arguments
Environment
Event Viewer
Select target
Select the manufacturer, board, project type and debug operation to use. Currently selected: Generic - MobiCore_Android
Filter platforms
Marvell PJ4 (TrustZone)
Mindspeed M84000 (Transcede 4000)
MobiCore_Android
Bare Metal Debug
Debug a <t-base Trusted Application
DS-5 Debugger will connect to <t-base Debug Server to debug a trustlet.
Connections
<t-base Trusted Application Debug
Trusted Application UUID
04010000000000000000000000000000
TA Resource File
CDB://trusted_app.rvc
Server Port
3010
Server Address
tcp:127.0.0.1
Browse...
Apply
Revert
Close
Debug


Debug Configurations
Create, manage, and run configurations
Create, edit or choose a configuration to launch a DS-5 debugging session.
type filter text
C/C++ Application
C/C++ Attach to Application
C/C++ Postmortem Debugger
C/C++ Remote Application
DS-5 Debugger
Trusted_App
Iron Python Run
Iron Python unittest
Java Applet
Java Application
JUnit
Jython run
Jython unittest
Launch Group
PyDev Django
PyDev Google App Run
Python Run
Python unittest
Remote Java Application
Filter matched 20 of 20 items
Name: Trusted_App
Connection
Files
Debugger
Arguments
Environment
Event Viewer
Target Configuration
Application on host to download:
File System...
Workspace...
Load symbols
Files
Load symbols from file
/home/SDK/t-sdk-r4/Samples/Rot13/TlSampleRot13/Out/Bin/Debug/tlSampleRot13.axf
File System...
Workspace...
Apply
Revert
Close
Debug



Arguments tab

- Use default. It should be empty

Environment tab

- Use default. It should be empty.

Event Viewer tab

- Use default. It should be empty

5.5.3.2.2 Debug Agent setup on device

- Verify if the <t-base debug driver (''07030000000000000000000000000000.tlbin') is present on the device in `/data/app/mcRegistry`.
- Make sure that `libMcTlDbg.so` is available on device (e.g. under /data/app).
- Check if the `com.rtec.tldebug` application is launched, check with "ps" command
  - If the `com.rtec.tldebug` application is not launched, please use the following command

```
adb shell am start -n com.rtec.tldebug/.TldebugActivity
```

- If you receive the following error message while launching `com.rtec.tldebug`, please wait some seconds and try again
- Before starting to debug, you should run the script `prepare_debug.sh`

- This script configures the Debug Agent to debug the Trusted Application chosen by the parameter given to the script
- The script accepts the following parameters: Rsa, Rot13, Aes, Sha256 or a UUID number
- If no parameter is passed, the script will simply perform the TCP forwarding via adb command
-

- In the Debug Configurations, start the debug procedure by pressing the Debug button. Now you should get a window indicating that the debugger tried to connect to the board.
- Go to the shell in the device (started by adb shell). Start the Trusted Application connector which loads the Trusted Application. Now the debugger should connect to the board (see also the chapters below).

```
Error type 2
android.util.AndroidException: Can't connect to activity manager; is the
system running?
```

If you receive `error: protocol fault (no status)` while launching `adb shell am start -n com.rtec.tldebug/.TldebugActivity`, then you probably are not connected to the device. Check this with the command `adb devices`.

5.5.3.3 Debugging a Trusted Application

- After setting up the environment, you need to build and push your Trusted Application binary to device
- While building your Trusted Application, you need to make sure that debug flag is set while building your Trusted Application. Meaning that bit # 3 is set to '1'. E.g.
  TRUSTLET_FLAGS := 4
- Load your Trusted Application. This is done by executing the Trusted Application connector 'application' in the normal world. During the creation of a session the Trusted Application is loaded into the secure world.
- Go to DS-5 '`Debug Control`' view. Select target and then '`Connect to Target`'
- Once DS-5 connection succeeds, you can debug your Trusted Application

5.5.3.4 Error Cases

- If the initial connection to the TA fails, make sure permissions for /dev/mobicore-user are set to 666 as the app does not run as root

6 TRUSTED APPLICATION CONNECTOR DEVELOPMENT

The following chapter describes how to set up your development PC appropriately and gives you the basis of TLC development using the Android NDK tool chain.

Building a TLC depends a lot on the platform you are developing for.

In the case of Android, the Native Development Kit (NDK) provided by Google is best suited to compile and build executable, shared- and static libraries.

Although it is possible to directly use the <t-base Driver API from your Android App, allowing you to write your TLC logic purely in Java code, this chapter focuses on developing a TLC in C code, so you are able to generate portable code for different platforms.

6.1 TRUSTED APPLICATION CONNECTOR STRUCTURE

The Trusted Application Connector is responsible for connecting the Android App with the Trusted Application. It provides the interface to the Trusted Application functionality on the Java layer.

Therefore a TLC is composed of the following components:

- One or more generic TLC methods using the <t-base driver interface to communicate with the Trusted Application in the Secure-World.
- A shared library written in C/C++ wrapping the generic TLC methods and providing their functionality to the Java layer.
- A Java class including the shared library and calling the library methods using Java Native Interface (JNI).
- An Android App including the Java class and the shared library.

The subsequent sections will provide a more detailed view on how to create these components.

6.2 COMPILE A TLC WITH THE ANDROID NDK

The Android NDK is used to build the Normal-World's native code, namely the TLC consisting of the shared library and an (optional) binary for testing purposes.

6.2.1 Android NDK Overview

The NDK is a stripped down version of the Android-Kernel tree. It comes equipped with the following components:

- The ARM cross compiling development tool chain
- The stable Android API's for native code development
- Android's Bionic C library, providing lightweight wrappers around kernel facilities
- A static library of the GNU libstdc++

Hint: Bionic is not binary-compatible with any other Linux C library. This means that you cannot build something against the GNU C Library headers and expect it to dynamically link properly to Bionic later.

More detailed information can be found in the "docs" folder of the NDK.

The NDK build system is based on Make, but encapsulates the generic build settings internally. It has been designed to meet the following goals:

1. Simple build files to describe the project. Basically you only need to list the C/C++ files and include directories.
2. The Android tool chain deals with many important details:
   - Proper tool chain selection and invocation (compiler + linker flags)
   - Android API level / platform / project support
   - Multi-ABI code generation
   - Native debugging setup

You need to define two Make files, which may reside in your projects "code" base folder:

- **Application.mk**. This is the main build file defining the modules required to build the project, which is NDK's naming for Make targets. You may define global variables here and need to point to the second Makefile.
- **Android.mk**. Here you need to define what each module is depending on:
  - Path to header file folders
  - Path to source files
  - (external) libraries
  - Type of output (at the end of a module's section):
    - Shared libraries:

```
include $(BUILD_SHARED_LIBRARY)
```

    - Static libraries:

```
include $(BUILD_STATIC_LIBRARY)
```

    - Executables:

```
include $(BUILD_EXECUTABLE)
```

See the best practice samples or the NDK's "docs" folder for more details on writing Make files.

Once you defined the Make files, the build can be invoked calling:

```
$ <path-to-ndk>/ndk-build \
           -B \
           -C <path-to-project> \
           NDK_DEBUG=1 \
           NDK_PROJECT_PATH=<path-to-project> \
           NDK_APPLICATION_MK=Application.mk \
           NDK_MODULE_PATH=<path-to-libMcClient.so> \
           NDK_APP_OUT=<path-to-output-dir> \
           APP_BUILD_SCRIPT=Android.mk
```

For a detailed description of the parameters, have a look at `documentation.html` in your NDK's base folder.

If you need to debug your build setup, add "V=1" as parameter for verbose output.

After a successful build the binaries can be found in `<path-to-output-dir>/local/armeabi/`.
If you defined an executable as one of your build targets, you can now push it from there to the device's `/data/app/` folder and execute it using ADB.

6.3 RUN YOUR TRUSTED APPLICATION CONNECTOR

6.3.1 Connect to the Device via USB

Connect your development device via USB and check with ADB if you can see the device:

```
$ adb devices
List of devices attached
3532C8CB5C0A00EC device
```

If you don't see a device ID, but:

```
List of devices attached
???????????    no permissions
```

You are missing permissions. Restart ADB with root privileges:

```
$ adb kill-server && sudo <path_to_SDK>/platform-tools/adb devices
```

6.3.2 Connect to the Device via Ethernet

Depending on your device, these may be required each time you restart the device. In case you connect the device directly to the host via USB, you can skip to the last point.

1. Obtain the IP address of the target. It can be found by adding "ip=dhcp" in the bootargs, which will obtain and print the IP automatically during boot. Alternatively, you can enable the Ethernet port and obtain an IP address via DHCP running the following commands in a terminal connected to the device once the platform has booted:

```
# netcfg eth0 up
# netcfg eth0 dhcp
```

2. Using the command below you can verify that the board did obtain an IP address:

```
# netcfg
```

3. On the host, perform the following (every time you reboot the device or create a new connection):

```
$ adb kill-server
$ adb connect <ip-address-of-device>:5555
```

4. Ensure that connection is working by running:

```
$ adb shell
```

You should see a command prompt of the target on your host.

Verify this by running `ps` or similar commands.

Exit the ADB shell by typing `exit`.

Other useful ADB commands are:

$ adb logcat	displays logging output of Android programs.
$ adb devices	shows available connected devices
$ adb shell	connects a shell to the device
$ adb shell <command>	executes a command on the device
$ adb connect <ip>:<port>	connects to a network connected device. Default port is 5555
$ adb kill-server	restarts adb server (on client) in case of problems (e.g. when getting "error: device offline")
$ adb push/pull <src> <dest>	uploads/downloads files to/from the device
$ adb install <file.apk>	installs an application to the device

Tab. 3 ADB commands

For a detailed description of all ADB commands see http://developer.android.com/tools/help/adb.html

6.3.3 Upload and Test

In order to test your TLC make sure the <t-base Driver Kernel module is loaded and the <t-base Driver daemon is running:

1. If not done yet, set up an ADB connection to the device:

```
$ adb connect <ip>:5555
```

Now you can upload and test your Trusted Application and TLC as described below:

2. Upload the Trusted Application to the device:

```
$ adb push <path-to-tl>/<your-tl> /data/app/mcRegistry
```

3. Upload and run the TLC:

```
$ adb push <path-to-tlc>/<your-tlc> /data/app/
$ adb shell /data/app/<your-tlc>
```

Depending on your TLC you should now see your debugging messages in the command shell.

6.3.4 Shared Libraries on the Device

By default the Android linker will search in /system/lib for shared libraries. In order to provide your own libraries for applications you need either make sure that you can write to this folder (root the device and remount the partition) or extend the LD_LIBRARY_PATH variable with a directory with sufficient access rights (e.g. /data/app).

The following command extends the library path:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/data/app
```

You need to set this each time the platform is restarted or the ADB shell is reconnected.

6.3.5 Trusted Application Connector usage from within your Android App

For the Java part of your development we recommend using Eclipse in connection with the Android Development Tools plugin.

Please refer to http://developer.android.com/guide/index.html for more information on developing the Java part of an Android App.

Good places to start are also the tutorials and sample code available at http://developer.android.com/training/index.html.

For the Android App to utilize the TLC's functionality it needs to import the shared library you created with the Android NDK. See the documentation and samples on http://developer.android.com/tools/sdk/ndk/index.html for a guideline on how to do that.

6.4 DEBUG

This section gives you an overview of all relevant debugging resources and strategies available on an Android device.

Below you can find a summary of some useful debugging commands for Linux system debugging to be executed from an Android shell:

Command	Description
dmesg	Displays Kernel debugging messages (kernel module output).
cat /proc/interrupts	Shows number of interrupts and their associated kernel module.
cat /proc/version	Displays the current Kernel version, when, from whom and which tool chain was used to build it.
ps (-t)	Prints current running processes (incl. threads if parameter "-t" is set) with their process ID (PID) and current program counter (PC) position.
cat /proc/''<PID>''/maps	Investigates memory mapping of a process.
cat /proc/meminfo	Shows current memory usage.
cat /proc/misc	Lists miscellaneous drivers registered on the miscellaneous major device.
lsmod	Lists all modules loaded into the kernel.
objdump -x <binary> | grep NEEDED	List the required libraries of a binary or library. The libraries must be found under /system/lib directory on the device.

Tab. 4 Linux system debugging commands

6.4.1 Segmentation Faults

In case you see something like this on the command line:

```
[1]   Segmentation fault      /data/app/tlcSampleRot13
```

Have a look at the output of logcat, e.g.:

```
I/DEBUG   (  678): *** *** *** *** *** *** *** *** *** *** *** *** *** ***
*** *** ***
I/DEBUG                  (            678):      Build      fingerprint:
'generic/generic/generic/:2.2/MASTER/eng.robert.20100629.090756:eng/test-
keys'
I/DEBUG   (  678): pid: 2071, tid: 2071  >>> /data/app/tlcSampleRot13<<<
I/DEBUG   (  678): signal 11 (SIGSEGV), fault addr 00000000
I/DEBUG   (  678):  r0 0001a4e4  r1 be882ca8  r2 00000003  r3 00000000
I/DEBUG   (  678):  r4 00018670  r5 40009008  r6 00000000  r7 00000000
I/DEBUG   (  678):  r8 00000000  r9 00000000  10 00000000  fp 00000000
I/DEBUG   (  678):  ip 00018708  sp be882ca0  lr 00012401  pc 00012404
cpsr 00000030
I/DEBUG        (     678):                          #00     pc   00012404
/system/data/app/tlcSampleRot13
I/DEBUG        (     678):                          #01     lr   00012401
/system/data/app/tlcSampleRot13
```

The interesting line is:

```
I/DEBUG        (     678):                          #00     pc   00012404
/system/data/app/tlcSampleRot13
```

It displays the binary/library which created the segfault and the content of the PC.

6.4.2 GDB

For debugging your C/C++ TLC code, the GDB (Gnu DeBugger) can be used. The setup is based on a gdbserver running on the device and an ARM aware GDB (arm-eabi-gdb) on the host machine.

While only the gdbserver and the arm-eabi-gdb binaries are required to debug native code on Android, you can also integrate debugging with Eclipse. The following section describes a method of debugging the NDK based applications from within Eclipse.

6.4.2.1 Debug stand-alone binaries

This section describes how to debug a stand-alone TLC binary (without the Java app on top).

For full GDB debugging functionality use Android 2.3 or later as your debugging platform.

ADB and device configuration:

1. set up an ADB connection to the device:

```
$ adb connect <ip>:5555
```

2. forward port 5039 to port 5039 on the device:

```
$ adb forward tcp:5039 tcp:5039
```

3. start GDB server with the program to debug on the device:

```
$ adb shell gdbserver :5039 /path/to/program
```

A detailed description to the last command can be found on http://www.kandroid.org/online-pdk/guide/debugging_gdb.html

Eclipse configuration:

First of all, Eclipse needs to find gdb:

- Make sure the Android SDK tools and the NDK binary folder are in your system path. For Linux, add to /home/<user>/.profile:

```
PATH="$PATH:<path-to-NDK>/build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/"
```

Configure a new `C/C++ Application` Debug configuration for your project.

Go to `Debug-View`, then `Debug` Button, then configure a new `C/C++ Application` Debug configuration for your project:

1. Main tab
   a. `C/C++ Application` must point to your binary.
2. Select debugger
   a. On the bottom: `Using GDB (DSF) Standard Create Process Launcher` → Select other
      i. Use GDB (DSF) **Remote system** Process Launcher
      ii. This will change existing tabs
3. Debugger tab
   a. `Stop on startup at:` can be set to your entry method (e.g. "main")
   b. In the subtab `Main` set `GDB debugger` to the `arm-eabi-gdb`
   c. In the subtab `Connection` set TCP, localhost and 5039

You need to restart the gdbserver after each debug run.

7 MOBICONVERT MANUAL

MobiConvert takes a service executable image (ELF file of a Trusted Application or driver) <in-file> and creates a <t-Base loadable service image <out-file>. Depending on the service type, it is either signed with a private key contained in a PEM format file or protected with a symmetric key stored in XML format file.

On success, a converted loadable image is created and the exit code is 0.

7.1 COMMAND-LINE INTERFACE

```
java -jar MobiConvert.jar <command>
<command>
-?, --help <num> Show help
no argument: show all available parameters
1: Show the help for service type 1,
2: Show the help for service type 2,
3: Show the help for service type 3,
4: Show the help for the header mode.
-h, --header <file> Show the header of the file given as argument
-s, --servicetype <num> <convoptions> The service type of the service to be converted
1: Driver,
2: Service Provider Trusted Application,
3: System Trusted Application. Conversion options, see below.
-g, --genuuid4 Generate a version 4 UUID according to RFC 4122 and exit
```

Note:
1.) The UUID is given without dashes

2.) There is no separate parameter to specify a fixed UUID when converting a Trusted Application or Driver. The <UUID> is expected to be passed in the filename of TLBIN with the -o parameter: `-o <UUID>.tlbin` (see also examples below)

<Convoptions> Parameter:

```
<convoptions>
For service type 1 (-s 1)
-b,--bin <in-file> -k,--keyfile <keyfile.pem>, -d,--driverid <id>, -o,--out
<pathToOutput>,
-iv,--interfaceversion <major.minor>, [-i,--numberofinstances <numberOfInstances>],
[-m,--memtype <memoryType>], [-n,--numberofthreads <numberOfThreads>], [-f,--flags
<flags>]
For service type 2 or 3 (-s 2 or -s 3)
-b,--bin <in-file>, -o,--out <pathToOutput>, -k,--keyfile <keyFile>,
[-f,--flags <flags>], [-m,--memtype <memoryType>], [-i,--numberofinstances
<numberOfInstances>], [-n,--numberofthreads <numberOfThreads
```

Detailed description:

```
-b, --bin <in-file> The path to the ELF input file
-d, --driverid <id> Driver Id, 31-bit unsigned integer (mandatory for
service type 1). The driver id must be > 100.
-f, --flags <flags> Flags (default = 0)
0: No flag,
1: Loaded service cannot be unloaded from
<t-Base,
```

```
2: Service has no WSM control interface,
4: Service can be debugged.
-iv, --interfaceversion <major.minor> The interface version to be used (mandatory for
service type 1). The range of major and minor shall
be each between 0-65535.
-k, --keyfile <keyfile> Key file to be used for the conversion.
<key>.pem containing the RSA keypair for
service type 1 or 3
<key>.xml containing the symmetric key in
its root element named "Key" for service type 2
-o, --out <out-file> Converted service file name. Filename must be of
form
<UUID>.drbin for service type 1
<UUID>.tlbin for service type 2 and 3
-m, --memtype, -m <type> The type of memory that should be used for the
service. (default = 2)
0: Internal memory prefered. If available use
internal memory, otherwise use external memory.
1: Use internal memory,
2: Use external memory.
-i, -numberofinstances <num> Max. number of concurrently active instances
(default = 1)
1: service type 1
1-16: for service 2 and 3
-n, --numberofthreads, -n <num> Max. number of concurrently active threads per
instance (default = 1)
1: service type 2 and 3
1-8: service type 1
```

Bitmask for flags is as follows:

	2	1	0
Permanent	**x**	**x**	**1**
No WSM (TCI/DCI)	**x**	**1**	**x**
Debuggable	**1**	**x**	**x**

7.2 CONVERSION FEATURES

7.2.1 Driver conversion

An ELF file is converted to a Driver (service type 1) with an asymmetric key that should be written with the PEM format.

A driver id and an interface version are mandatory parameters.

The output is then the Driver) called <uuid>.drbin.

Usage:

```
java -jar MobiConvert.jar -b <pathToBinary> -o <pathToOutput> -k <pathToKeyFile> -s
<serviceType>
-d <driverId> -iv <major.minor> [-i <numberOfInstances>] [-n <numberOfThreads>] [-m
<memoryType>]
[-f <flags>]
```

7.2.2 Service Provider Trusted Application conversion

An ELF file is converted to a Service Provider Trusted Application (service type 2) with a symmetric AES key provided in an XML file.

The output is the Service Provider called <uuid>.tlbin.

Usage:

```
java -jar MobiConvert.jar -b <pathToBinary> -o <pathToOutput> -k <pathToSymmetricKeyFile>
-s <serviceType> [-m <memoryType>] [-i <numberOfInstances>] [-n <numberOfThreads>] [-f
<flags>]
```

7.2.3 System Trusted Application conversion

An ELF file is converted to a System Trusted Application (service type 3) with a asymmetric key provided in the PEM format.

The output is the Service Provider called <uuid>.tlbin.

Usage :

```
java -jar MobiConvert.jar -b <pathToBinary> -o <pathToOutput> -k <pathToKeyFile> -s
<serviceType>
[-f <flags>] [-m <memoryType>] [-i <numberOfInstances>] [-n <numberOfThreads>]
```

7.2.4 Header mode

The header mode provides a way to display the content of the header of a service that was converted.

The name of the file provided as argument does not need to respect the "uuid.tlbin/drbin" format.

Usage :

```
java -jar MobiConvert.jar -header <file>
```

7.2.5 Examples

7.2.5.1 Converting a Driver

```
java -jar MobiConvert.jar -s 1 -b drSample.axf -k myKey.pem -driverid 102 -iv 1.0
-o 15010000100000000000000000000000.drbin
```

7.2.5.2 Converting a Service Provider Trusted Application

```
java -jar MobiConvert.jar -s 2 -b tlSample.axf -k mySymmetricKey.xml
-o 15010000100000000000000000000000.tlbin
```

7.2.5.3 Converting a Service Provider Trusted Application with a debuggable flag

```
java -jar MobiConvert.jar -s 2 -b tlSample.axf -k mySymmetricKey.xml -f 4
-o 15010000100000000000000000000000.tlbin
```

7.2.5.4 Converting a Service Provider Trusted Application with a debuggable and a permanent flag

```
java -jar MobiConvert.jar -s 2 -b tlSample.axf -k mySymmetricKey.xml -f 5
-o 15010000100000000000000000000000.tlbin
```

7.2.5.5 Converting a System Trusted Application

```
java -jar MobiConvert.jar -s 3 -b tlSample.axf -k myKey.pem
-o 15010000100000000000000000000000.tlbin
```

7.2.5.6 Using the Header mode

```
java -jar MobiConvert.jar -h tlSample.tlbin
```