TRUSTONIC

DEVELOPER'S GUIDE

‹t-base
Developer's Guide

PREFACE

VERSION HISTORY

Version	Date	Modification
1.0	July 18th, 2013	First version
2.0	November 18th, 2013	Updated for <t-base-300
2.1	January 13th, 2014	Clarifications added
2.2	April 4th, 2014	SPPA for OTA provisioning added
2.3	May 13th, 2014	Clarifications added for API availability
2.4	June 9th, 2014	Updated for <t-base-301
2.5	August 25th, 2014	Updated for <t-base-301B - Update MobiConvert Manual - Explain usage of UUID attestion for GP TAs. - Clarify stack and heap constraints

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

This Developer's Guide is a practical introduction for developing Trusted Applications for <t-base and their corresponding Client Applications.

This guide provides an overview of the different sets of API available and explains how to use the <t-sdk.

The full <t-base API specification can be found in the `<t-base API Documentation`.

1.1 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.
CA	**C**lient **A**pplication. Software running in the NWd providing a high level convenience interface to access a Trusted Application in the SWd by the NWd client.
DMA	**D**irect **M**emory **A**ccess
DS-5™	ARM **D**evelopment **S**tudio 5
GDB	**G**nu **De**Bugger
GNU	**G**NU's **N**ot **U**nix
GP	**G**lobal **P**latform
IPC	**I**nter-**P**rocess **C**ommunication, communication between two tasks running in different processes.
IWC	**I**nter-**W**orld **C**ommunication, communication between two tasks running in different worlds.
JNI	**J**ava **N**ative **I**nterface
JTAG	**J**oint **T**est **A**ction **G**roup
JTAG DCC	JTAG **D**igital **C**ommand **C**ontrol
<t-base Ecosystem	The whole infrastructure enabling the deployment of <t-base enabled devices as well as the installation and operation of security protected applications using the <t-base Operating System.
MMU	**M**emory **M**apping **U**nit

Multi ABI	Multiple **A**pplication **B**inary **I**nterface
NDK	Android **N**ative **D**evelopment **K**it
NWd	**N**ormal **W**orl**d**. The software system running on an ARM TrustZone enabled device in non-secure mode.
OS	**O**perating **S**ystem
OTA	**O**ver-**T**he-**A**ir
PC	**P**rogram **C**ounter
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
SP-App	**S**ervice **P**rovider **App**lication: This is the implementation of the service function provided by the Service Provider (SP) to the end-user. The SP-app code is provided by the service provider. It includes the user interface and all the business logic which shall be implemented on the device side. For security sensitive operations SP-App requires functions implemented in a Trusted Application (TA).
SPPA	**S**ervice **P**rovider **P**rovisioning **A**gent:
<t-sdk	<t-base **S**oftware **D**evelopment **K**it
SWd	**S**ecure **W**orld, software system running in secure mode in the ARM TrustZone
TA	**T**rusted **A**pplication on the SWd side executing specific security services in the <t-base runtime environment. The security services provided by a Trusted Application are called by NWd client applications. <t-base differentiates between System Trusted Application like the CM Trusted Application and Trusted Applications of Service Providers.
TAM	**T**rusted **A**pplication **M**anager: online server communicating with the RootPA and SPPA to provision containers in a safely and secured way.
TEE	Trusted Execution Environment. <t-base is an instantiation of a TEE. The term TEE is defined in the OMTP specifications.
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 share memory. Shared memory which will be used for IPC between NWd and SWd.

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.

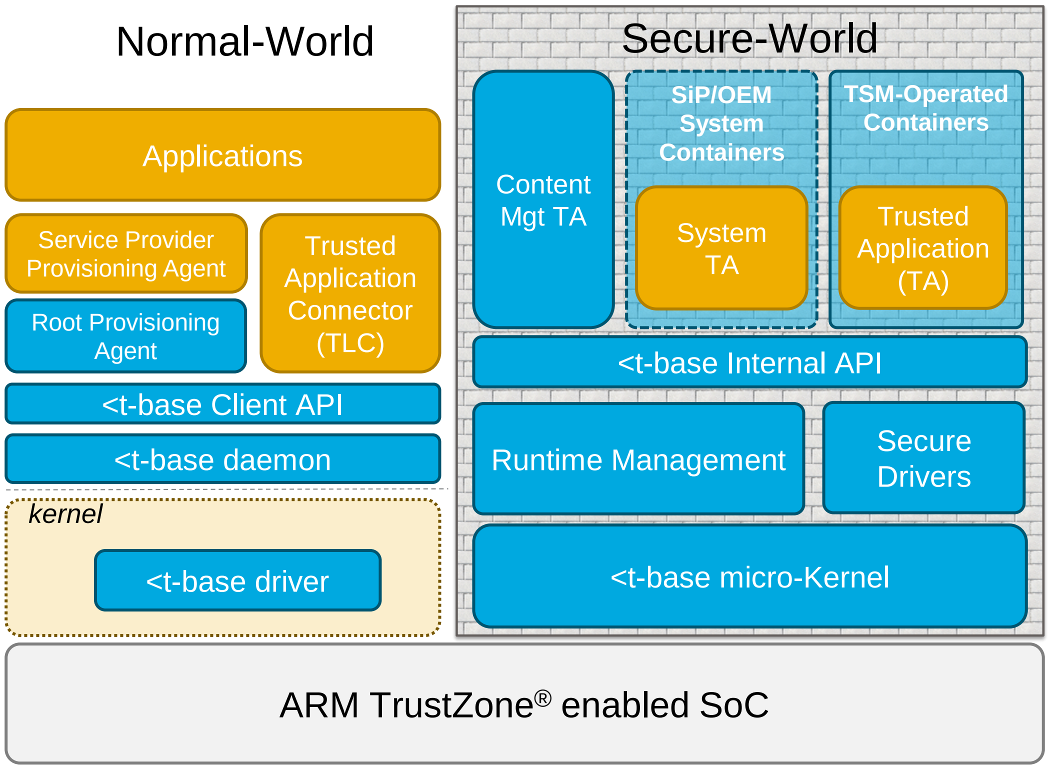


Figure 1: <t-base Architecture Overview.

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.

2.1 <T-BASE API FOR DEVELOPERS

<t-base is an open environment which provides API for developers to develop Trusted Applications and Client Applications. <t-base provides two distinct set of APIs for developers:

- The <t-base Legacy API to develop legacy Trusted Applications using the tlAPI and the mcClient API.
- The GlobalPlatform API to develop Trusted Applications as defined by GlobalPlatform using the GlobalPlatform Client and Internal APIs.

The <t-play, TUI or GP APIs may not be available on certain commercial devices.

The implementation of these features may not be possible due to hardware limitations or other constraints.

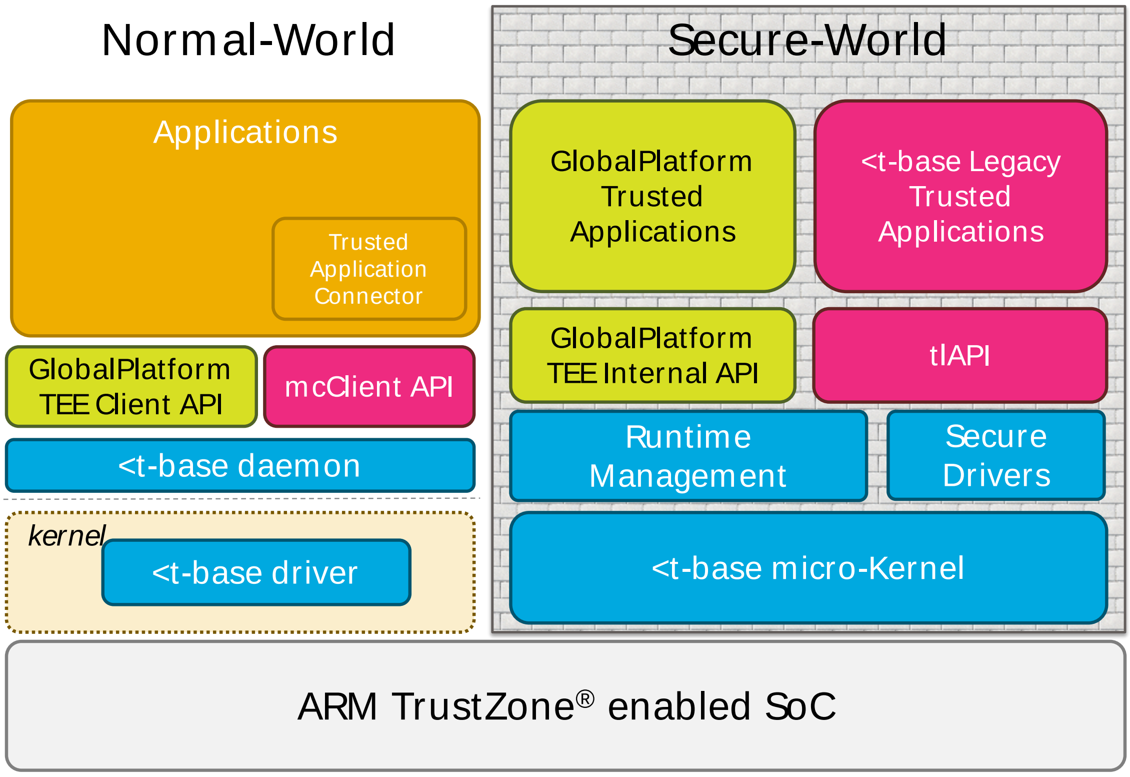

Normal-World
Secure-World
Applications
Trusted Application Connector
GlobalPlatform TEE Client API
mcClient API
<t-base daemon
kernel
<t-base driver
GlobalPlatform Trusted Applications
<t-base Legacy Trusted Applications
GlobalPlatform TEE Internal API
tlAPI
Runtime Management
Secure Drivers
<t-base micro-Kernel
ARM TrustZone® enabled SoC

Figure 2: <t-base API Overview.

3 Development Guidelines

A rich operating system cannot provide the trustworthiness that is needed to keep up with the ongoing improvements of attacks and threats, coming with new functionalities, on its own. This means that security mechanisms need to be integrated in the system on 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 GENERAL GUIDELINES

3.1.1 Defining the scope of the Trusted Application

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

The prerequisite for implementing a Trusted Application and Client Application 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 securely.

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 security 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 picks up 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 NWd to eavesdrop on the user secrets.

After the split into secure and non-secure part is decided, the application designer defines the state in the Trusted Application and the interface between the Trusted Application and the Client Application. The design must take into consideration the restrictions of the TA environment and the CA-TA interfaces.

3.1.2 Trusted Application Design

Developers should also not the following generic guidelines:

- Separate your Trusted Applications functionality into multiple operations, 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, they can only use the Trusted Application API.
- Never write security critical data to the World-Share Memory as it might be read by the Normal-World at any time.

3.1.3 Trusted Application Address Space

In <t-base, Trusted Application can use a total of 1012kB for program data. The required memory will be allocated from secure memory when the TA is loaded. If the system does not have enough of the requested memory, loading fails.

For sharing data with the Client Application, <t-base provides a 5 MB area of ***World Shared Memory*** (WSM) in the TA address space. The first WSM is mapped when the session with the NWd Client Application is opened. The address of this WSM and its length is given to the Trusted Application entry function (`tlMain()`).

Then 4 additional slots of a maximum size of 1MB can be mapped on-demand.

Notice that 1MB is a maximum size when the WSM is aligned to 4KB.

Trusted Applications must be especially careful when handling data coming from the Client Application through World-Shared Memory buffers: the content of the buffers may change at any time, even between two consecutive memory accesses by the Trusted Application. This means that the Trusted Application should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Trusted Application should always read data only once from a World-Shared Memory buffer and then validate it. It must not assume that data written to the buffer can be read unchanged later on.

The Trusted Application ***stack*** is also part of the program data and is statically allocated during compile time. The developer has to use the following macro to declare the required stack size.

```
DECLARE_TRUSTED_APPLICATION_MAIN_STACK(4096);
```

A Trusted Application can use common ***heap*** functionality using `tlApiMalloc`, `tlApiFree` and `tlApiRealloc` to organize its dynamic memory. However, the heap is also allocated during compile time and part of the TA program data. The developer has to use the following macro to reserve the required heap size.

```
DECLARE_TRUSTED_APPLICATION_MAIN_HEAP(16384);
```

Note that there is no overflow protection for the Trusted Application stack. Big data structures and arrays should be defined as global variables instead of as local variables. When a developer defines local variables in a function, the C language will allocate these variables on the stack. When such variables contain big data structures and arrays, the developer has to make sure that the stack is at least as big as the data structures. Otherwise, modifying a local variable can result in corrupting global variables and the heap.

See Figure 3 for the layout of the virtual address space of a TA:

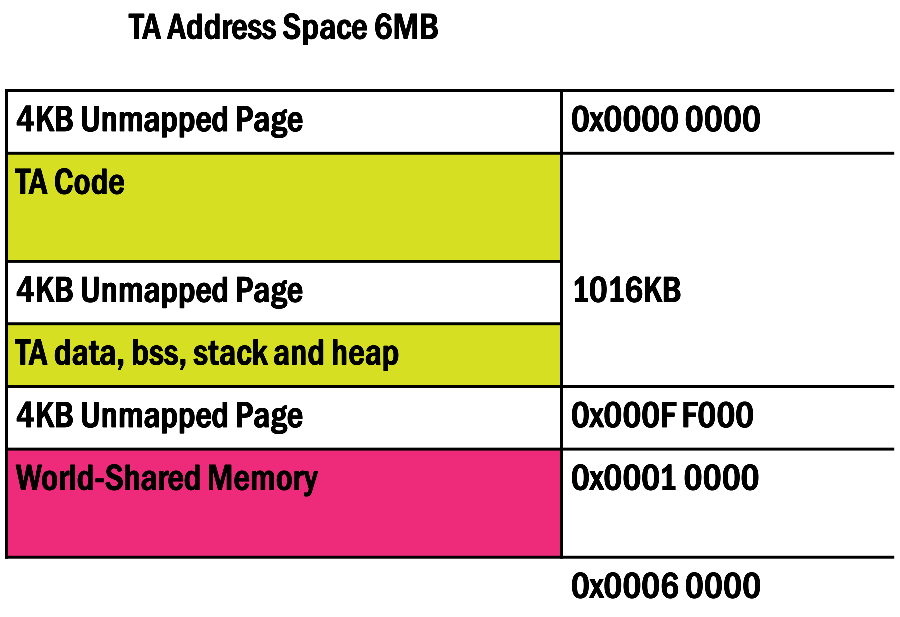


Figure 3: TA Virtual Address Space.

3.2 USING THE LEGACY API

3.2.1 Client Applications

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.

3.2.1.1 Trusted Application Connector

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 mcClient 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

3.2.1.2 Using the mcClient API

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

3.2.1.2.1 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.

3.2.1.2.2 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 mcOpenTrustlet(
    mcSessionHandle_t  *session,
    mcSpid_t           spid,
    uint8_t            *trusted_application,
    uint32_t           tLen,
    uint8_t            *tci,
    uint32_t           tciLen
);
```

`mcOpenTrustlet()` 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.

3.2.1.2.3 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 the <t-base API Documentation for further details.

3.2.1.2.4 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. 3 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.

3.2.1.3 Client Kernel API

The mcClient API is also available at the kernel level and can be called by kernel modules. The mcClient API functions are the same in user mode or in kernel mode.

3.2.2 Trusted Applications

3.2.2.1 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.

3.2.2.2 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();
    }
}
```

3.2.2.3 Security Considerations

Trusted Applications must be especially careful when handling data coming from the Client Application through World-Shared Memory buffers: the content of the buffers may change at any time, even between two consecutive memory accesses by the Trusted Application. This means that the Trusted Application should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Trusted Application should always read data only once from a World-Shared Memory buffer and then validate it. It must not assume that data written to the buffer can be read unchanged later on.

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!

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.

3.2.2.4 <t-base tlAPI

The <t-base tlAPI is used by Legacy 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 the <t-base API Documentation for complete reference.

3.2.2.5 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 tlApiSpTrusted ApplicationId_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 tlApiSpTrusted ApplicationId_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.

3.2.2.6 Endorsement API

<t-base provides an API which allows a Trusted Application to sign data and prove that it is generated in a genuine TEE and in the right Trusted Application. This feature is important for service providers such as network operators.

The data is signed in <t-base with a key derived from the Hardware Unique Key and is encrypted with Trustonic public endorsement key. The encrypted signed data can be exported to a server of the service provider and the service provider can then verify the signature through the Trustonic <t-directory server interface.

The function to generate such endorsement is tlApiEndorse() and is described in the <t-base API Documentation.

3.2.2.7 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.

3.2.3 Trusted User Interface

The Trusted User Interface (TUI) feature in <t-base allows a TEE to interact directly with the user via the common display and touch screen. <t-base temporarily protects those using TrustZone.

The main objective of the TUI is to protect the confidentiality and integrity of the information exchanged between a Trusted Application and the user against the NWd OS (Android).

This main objective is achieved by three features:

- Secure Input: The information entered by the user to a Trusted Application cannot be derived or modified by any software within the NWd OS or by another unauthorized Trusted Application.
- Secure Display: The information displayed by the Trusted Application cannot be accessed, modified, or obscured by any software within the NWd OS or by another unauthorized Trusted Application. The secure display should be completed by a Secure Indicator.
- Security Indicator: The Trusted Application securely displays a secret, previously shared with the user, making the user confident that the screen displayed is actually displayed by a Trusted Application.

The TUI targets quite simple use cases like message display, PIN or password entry. These use cases require exclusive access of UI resources, not permanently but for a short period of time. This period is called a TUI session.

Within a TUI session, the SWd must have the full control of the UI resources. Thus the NWd drivers are suspended at the beginning of the session, and resumed after the session. Particularly the NWd must not be aware of any input event within the session.

A TUI session may be cancelled in case of some system events:

- Reset or turn off the device,

- Screen off,
- Incoming call

Trusted Applications can use the Trusted User Interface through a simple API which let them display data on the screen and get user inputs.

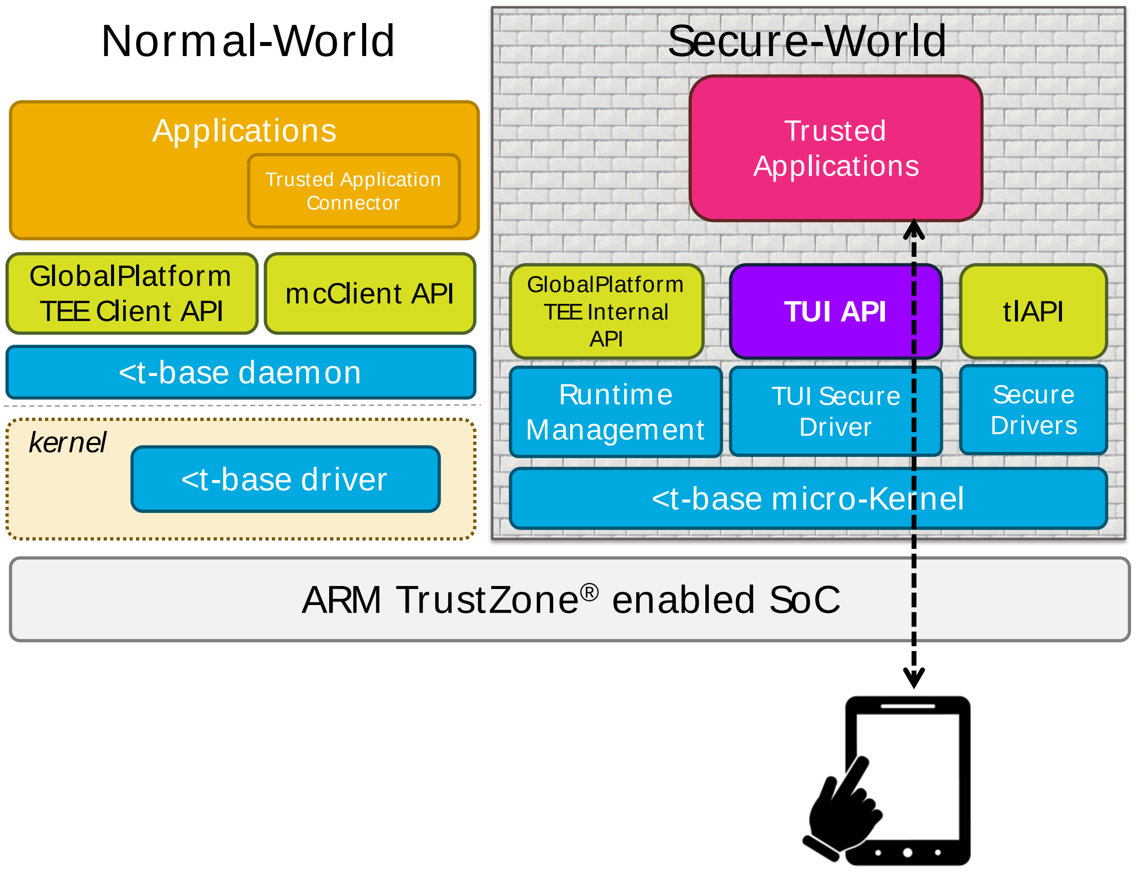


Figure 4: Trusted User Interface API.

3.2.3.1 TUI configuration

A Trusted Application that uses TUI is responsible for its own display layout. It may be composed knowing the touchscreen metrics.

```
typedef struct {
    uint32_t    grayscaleBitDepth;
    uint32_t    redBitDepth;
    uint32_t    greenBitDepth;
    uint32_t    blueBitDepth;
    uint32_t    width;
    uint32_t    height;
    uint32_t    wDensity;
    uint32_t    hDensity ;
} tlApiTuiScreenInfo_t, *tlApiTuiScreenInfo_ptr;

_TLAPI_EXTERN_C tlApiResult_t tlApiTuiGetScreenInfo (
      tlApiTuiScreenInfo_ptr screenInfo)
```

3.2.3.2 TUI Session

To use the TUI, a Trusted Application has to open a TUI session. When the Trusted Application has finished the user interaction, it must close the TUI session.

Except for the configuration, all TUI functions must be called within a TUI session.

```
tlApiResult_t tlApiTuiInitSession(
 void
);

tlApiResult_t tlApiTuiCloseSession(
 void
);
```

3.2.3.2.1 Normal world integration

The <t-base TUI handles the necessary Normal-World integration.

The TUI application-developer does not have to care about incoming phone calls, screen off or power down events in his Normal-World application. He can use a regular TLC to communicate in the usual way with the Trusted Application. Only the Trusted Application interacts with the TUI feature using the TlApiTui functions.

When the Trusted Application starts the TUI session, <t-base takes control over the display. For the TLC it is as if another Android activity occupies the screen for the duration of the TUI session.

3.2.3.2.2 Error code checking

The Trusted UI session can be interrupted at any time by phone calls and power management events. Hence the Trusted Application cannot expect to complete the user interaction every time. The developer has to foresee restarting the interaction and possibly saving and restoring interaction state. As a consequence, every TlApiTui call that is associated with the TUI session can fail with an error code indicating that the Trusted Application no longer has a TUI session. It is therefore mandatory that the Trusted Application always checks the return values of all tlApiTui functions.

```
#define E_TLAPI_TUI_NO_SESSION      0x00000501
```

3.2.3.2.3 Single user service

Only one Trusted Application can use the Trusted UI at a time and it controls the whole screen. If a Trusted Application tries to open a TUI session while another Trusted Application already uses it, an error code is returned.

```
#define E_TLAPI_TUI_BUSY            0x00000502
```

3.2.3.3 Secure display

To give the user visual information, a Trusted Application can draw image files on the secured display. <t-base supports PNG files.

```
tlApiResult_t tlApiTuiSetImage(
 [in] tlApiTuiImage_t *image,
 [in] tlApiTuiCoordinates_t coordinates
);
```

```
typedef struct {
    void*        imageFile;
    uint32_t     imageFileLength;
} tlApiTuiImage_t;
```

Using the coordinates, the position of the image can be defined. <t-base-will read the image header to find out about the width and height of the image. Requests to draw partially outside of the display will be rejected. The Trusted Application has to provide the image in a buffer in the Trusted Application address space.

Note on address space limitations and buffer sizes

Modern smartphone display resolutions range from 1024 x 600 pixels to 1920 x 1080 pixels, totaling in a frame buffer size of 2 MB to 6 MB. Tablets devices with screen resolutions of up to 2560 x 1600 pixels feature a frame buffer size of around 12 MB. PNG compression algorithms can reduce images with graphical content by a factor of 6 to 25. So a typical full-screen user interface should well fit into a 200 KB PNG image.

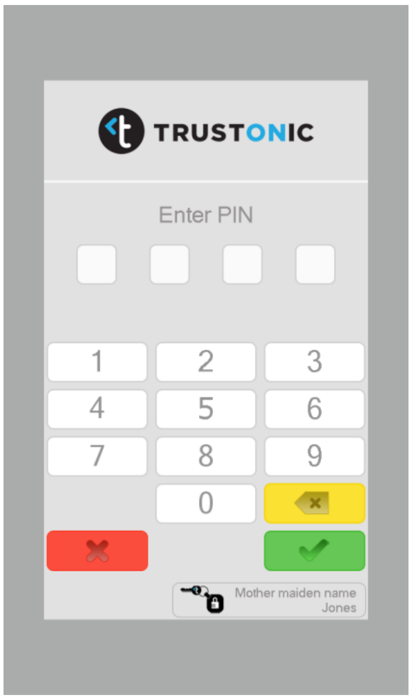


Fig. 1. PINpad sample.png 1024 x 600 at 32,6 KB

Note that the developer is responsible for protecting the resources to be displayed, such as the images.

3.2.3.4 Secure input

To receive trusted input from the user during a TUI session, the Trusted Application has to wait for a notification and then call the driver to get the touch event data.

```
tlApiResult_t tlApiTuiGetTouchEvent(
  [out] tlApiTuiTouchEvent_t *touchEvent
);

typedef struct {
    tlApiTuiTouchEventType_t   type;
    tlApiTuiCoordinates_t      coordinates;
} tlApiTuiTouchEvent_t;
```

```
typedef enum {
    TUI_TOUCH_EVENT_RELEASED = 0,
    TUI_TOUCH_EVENT_PRESSED  = 1,
} tlApiTuiTouchEventType_t;

typedef struct {
    uint32_t    xOffset;
    uint32_t    yOffset;
} tlApiTuiCoordinates_t;
```

3.2.3.4.1 Asynchronous driver interface

Trusted Applications in <t-base are already event driven, based on the notification and wait- for-notification primitives. These functions were primarily used to communicate with the TLC. For the Trusted User Interface, the Trusted Application can actually wait for an event from a TLC and for an event from a driver. That way, the Trusted Application remains single threaded, can receive abort or timeout commands from the TLC and, at the same time, wait for hardware-induced events like touches on a touch screen.

At the API level, Trusted Applications can call a function like tlApiGenerateKey() and this blocks the Trusted Application until the function returns. But the function flow can also be split into three parts:

- driverProgramOperation(),
- waitNotififcation,
- driverGetOperationResult()

For the Trusted User Interface, this means that once the Trusted Application has opened a TUI session, it receives notifications for touch events and can receive the event data with the `getTouchEvent()` functionality. Note also that any normal-world-induced abort event creates one last notification that also closes the TUI session.

3.2.3.4.2 Trusted Application state machine

The developer has to decide which parts of the application logic have to be protected through a Trusted Application. A Trusted User Interface is also a Graphical User Interface and should follow common design principles for GUIs. The developer may implement helper functionalities such as font management to manage UI components in addition to the business logic.

Since the `waitNotification()` function does not indicate the origin of the notification to the Trusted Application, the Trusted Application has to maintain state and act depending on what it expects in a certain situation. If the Trusted Application hasn’t enabled the Trusted UI session yet, there is no use in calling the `getTouchEvent()` function. If the TUI session is started, maybe certain commands from the TLC should be ignored. If the Trusted Application implements an input field, characters already entered and the current cursor position must be remembered. If the Trusted Application implements multiple dialogs it might want to track the dialog that is currently active.

3.2.3.4.3 Input filtering and button emulation

Modern touch controllers create many touch events per second and they are all send to the Trusted Application that has opened a Trusted UI session. It is up to the Trusted Application to apply relevant event filtering and cumulate hardware touch events to application logic input events. For every touch event, <t-base TUI holds the coordinates and the Trusted Application has to match those coordinates against a list of possible sensitive areas to determine a “button” press action.

3.2.4 DRM API

<t-base provides a DRM API for Trusted Applications to process DRM Content. This API is simple and hides all the complexity of data decryption, decoding and rendering to the system and Secure Driver.

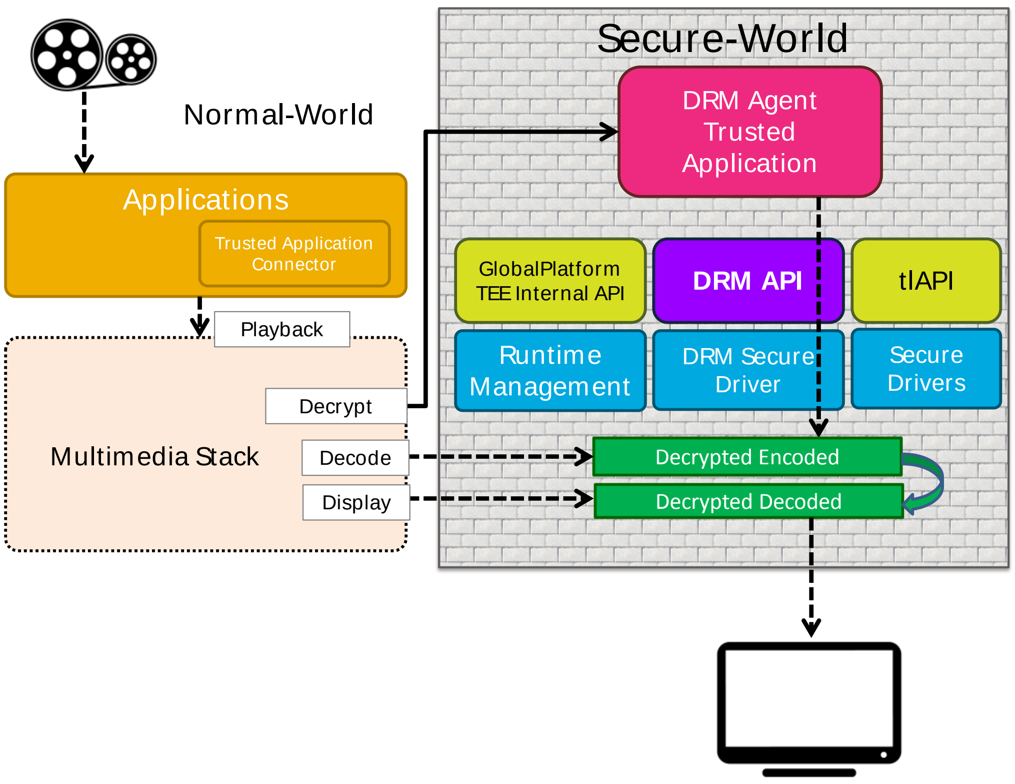


Figure 5: <t-base DRM API.

The DRM API is made of one main function, `tlApiDrmProcessContent()` which processes the encrypted content. The caller must have created a decryption context first with the appropriate key and parameters for the content decryption.

```
int32_t tlApiDrmProcessDrmContent (
    uint8_t                       sHandle,
    TL_DRM_DecryptContext         decryptCtx,
    uint8_t                      *input,
    TL_DRM_InputSegmentDescriptor inputDesc,
    uint16_t                      processMode,
    uint8_t                      *rfu);
```

3.3 USING THE GLOBALPLATFORM API

<t-base supports some of the APIs defined by GlobalPlatform to develop Client Applications and Trusted Applications.

Developers should refer to the GlobalPlatform specifications for details about the GlobalPlatform APIs and how to use these APIs:

- http://www.globalplatform.org/specificationsdevice.asp

This version of <t-base supports parts of the following specifications:

- TEE Client API Specification v1.0
- TEE Client API Specification v1.0 Errata and Precisions v1.0
- TEE Internal API Specification v1.0
- TEE Internal API Specification v1.0 Errata and Precisions v2.0

The TEE Client API is fully supported. The **<t-base API Documentation** details the implementation notes.

The <t-base TEE Internal API does not support all the features defined in GlobalPlatform and in particular does not support:

- Arithmetic, DSA and DH algorithms,
- TA to TA communication,
- Time API,
- Persistent Objects and Properties enumerations

Please refer to the **<t-base API Documentation** which lists exactly which functions are supported and details the implementation notes.

<t-sdk also contains examples developed with the GlobalPlatform APIs.

3.3.1 Client Applications

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

- Initiating a context to the TEE
- Opening and closing sessions to one or more Trusted Applications
- Registering and unregistering shared memory
- Invoking commands on sessions, passing command data and memory buffers to a Trusted Application
- Handling Trusted Application return codes and data returned either directly or in memory.

Operations

When a CA sends a command to a TA, the command contains a command identifier and an operation payload in accordance with the protocol that the TA exposes for that command. An operation contains 4 parameters that can be used to exchange data from the CA to the TA and from the TA to the CA. Parameters are typed and can either denote an immediate value or a reference to a memory region. The answer to a command invocation always contains a return code. Return codes are typed and give an indication as to where in the software stack a possible error occurred.

Cancellation

Calls to the TEE Client API are synchronous, meaning that a long-running operation in the TA blocks the CA. When a CA is reactive also to user input, network timeouts and other events, the CA should use multiple threads. If a thread of a CA is blocked by a TA, another thread of the CA can call the cancel function on this TA session, effectively stopping the TA session and unblocking the other CA thread.

This version of <t-base supports the cancellation of blocking API functions (TEEC_OpenSession and TEE_InvokeCommand).

3.3.2 Trusted Applications

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

3.3.2.1 TA Lifecycle

A Trusted Application is command-orientated. Client Applications access a service by opening a session with the Trusted Application and sending commands within the session. When the Trusted Application receives a command, it parses any message associated with the command, performs any required processing and then sends a response back to the Client Application.

When a Client Application opens a session with a Trusted Application, <t-base creates a new process and loads the TA binary into it and starts execution of this process. This process is then called an instance of that Trusted Application. Trusted Application processes are single-threaded and have their own address space separating them from all other processes.

In this version of <t-base, there is a one-to-one relationship between a Client Application's session and an instance of a Trusted Application: When a CA opens a session to a TA, the TA is started and accepts commands in this session. A TA processes one command at a time. When the CA closes the session, the TA will destroy itself. If a CA opens another session to an already open TA, a new independent instance of this TA is created.

Comparison with the TA lifecycle defined in the GP TEE API:

- All TAs are multi-instance TAs, there are no single-instance TAs.
- One TA instance can only have one session at a time, there is no multi-session TA.
- TA lifetime is linked to CA session, no support for keeping alive instances.
- No support for TA Client API, no communication between TAs

3.3.2.2 TA Interface

Each Trusted Application must implement a few callback functions, collectively called the "TA interface". These functions are the entry points called by the <t-base TEE to create the instance, notify the instance that a new client is connecting, notify the instance when the client invokes a command and to notify when a client is disconnecting and also before destroying the instance. These functions must be implemented in order to link the TA binary. The TA is single-threaded, all entry points are linked into the same address space and <t-base ensures that only one entry point is called at a time. When a TA returns from an entry point, the TA is either destroyed or blocked until the next entry point is called.

Table 1 lists the functions in the TA interface.

Table 2: TA Interface Functions

TA Interface Function	Description
TA_CreateEntryPoint	This is the Trusted Application constructor. It is called once and only once in the life-time of the Trusted Application instance. If this function fails, the instance is not created and the process associated

	with this Trusted Application is freed. After this function, <t-base always calls the TA_OpenSessionEntryPoint.
TA_OpenSessionEntryPoint	This function is called whenever a client attempts to connect to the Trusted Application instance to open a new session. If this function returns an error, the connection is rejected and no new session is opened. In this function, the Trusted Application can attach an opaque void* context to the session. This context is recalled in all subsequent TA calls within the session.
TA_InvokeCommandEntryPoint	This function is called whenever a client invokes a Trusted Application command. The <t-base TEE gives back the session context reference to the Trusted Application in this function call.
TA_CloseSessionEntryPoint	This function is called when the client closes a session and disconnects from the Trusted Application instance. The session context reference is given back to the Trusted Application by the <t-base TEE. It is the responsibility of the Trusted Application to deallocate the session context if memory has been allocated for it.
TA_DestroyEntryPoint	This is the Trusted Application destructor. The <t-base TEE calls this function just before the Trusted Application instance is terminated. The function is always called after the TA_CloseSessionEntryPoint. When TA_DestroyEntryPoint returns, <t-base destroys the process, frees all resources and notifies all secure drivers that the Trusted Application instance was destroyed.

Table 2 summarizes client operations and the resulting Trusted Application effect.

Table 3: Correspondence between CA operation and TA Interface

Client Operation	**Trusted Application effect**
TEEC_OpenSession	<t-base creates a new process and calls first the TA_CreateEntryPoint for the process to initialize and then the TA_OpenSessionEntryPoint.
TEEC_InvokeCommand	<t-base unblocks the process and calls TA_InvokeCommandEntryPoint.
TEEC_CloseSession	<t-base unblocks the process and calls TA_CloseSessionEntryPoint. Afterwards <t-base calls TA_DestroyEntryPoint and then destroys the process.
Client terminates unexpectedly	<t-base supports ordered shutdown of TA in case of CA crash. From the point of view of the TA instance, the behavior is identical to the situation where the CA requests the cancellation of all pending operations in that session, waits for the completion of all these

	operations in that session, and finally closes that session. Note that there is no way for the TA to distinguish between the client gracefully cancelling all its operations and closing all its sessions and the implementation taking over when the client dies unexpectedly.

3.3.2.3 TA Secure Storage

To keep state between instances <t-base has support for the GP persistent storage API. Using functions similar to fopen, fclose, read, write, and seek, Trusted Application can put object data into files. Those files are identified with a byte-string of up to 64 bytes. TAs cannot share files. Files or persistent objects are bound to one device and one TA. Only one instance of a TA can open a file at a time.

Persistent object are stored in secured files normal world file system by <t-base daemon. <t-base cannot prevent removal or modification of those files, but modifications will be detected.

This version of <t-base only supports relatively small files: The entire file must fit into the heap of the TA.

3.3.2.4 Implementing the TA Interface

A Trusted Application has to implement the TA Interface and can use the TEE Internal API. See the example code provided with the <t-sdk.

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

```
TEE_Result TA_EXPORT TA_CreateEntryPoint(void)
{
    return TEE_SUCCESS;
}

void TA_EXPORT TA_DestroyEntryPoint(void)
{
}

TEE_Result TA_EXPORT TA_OpenSessionEntryPoint(
                           uint32_t   nParamTypes,
                           INOUT      TEE_Param pParams[4],
                           OUT void** ppSessionContext)
{
    return TEE_SUCCESS;
}

void TA_EXPORT TA_CloseSessionEntryPoint(INOUT void* pSessionContext)
{
}

TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(
                           INOUT void* pSessionContext,
                           uint32_t    nCommandID,
                           uint32_t    nParamTypes,
                           TEE_Param   pParams[4])
{
    switch(nCommandID)
    {
```

```
    case CMD_HELLO:
        // do some processing
        return TEE_SUCCESS;
    }
}
```

In between the entry points, the <t-base TEE has code that runs when the TA starts and synchronizes with the CA. Every Trusted Application has to implement these functions, similar to the main() function in regular c-programs.

3.3.2.5 Implementing the Operations

To support a comprehensive communication protocol, a Trusted Application exports several commands and associated parameters. The TA_OpenSessionEntryPoint and the TA_InvokeCommandEntryPoint from the TA Interface define operation objects that can be used to communicate with the Client Application. An operation contains 4 parameters that have one of three possible directions: IN, OUT, INOUT. Parameters are typed and can either denote an immediate value or a reference to a memory region:

```
typedef union {
    struct {
        void*    buffer;
        size_t   size;
    } memref;
    struct
    {
        uint32_t a;
        uint32_t b;
    } value;
} TEE_Param;
```

The `nParamTypes` parameter defines the type and direction of the 4 parameters:

```
/* Parameter types */
#define TEE_PARAM_TYPE_NONE                     0x0
#define TEE_PARAM_TYPE_VALUE_INPUT              0x1
#define TEE_PARAM_TYPE_VALUE_OUTPUT             0x2
#define TEE_PARAM_TYPE_VALUE_INOUT              0x3
#define TEE_PARAM_TYPE_MEMREF_INPUT             0x5
#define TEE_PARAM_TYPE_MEMREF_OUTPUT            0x6
#define TEE_PARAM_TYPE_MEMREF_INOUT             0x7

#define TEE_PARAM_TYPES(t0,t1,t2,t3)  ((t0) | ((t1) << 4) | ((t2) << 8) |
((t3) << 12))
```

The macro `TEE_PARAM_TYPES` can be used to combine the 4 parameter definitions into the `nParamTypes` value.

The following sample code illustrates the usage:

```
TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(void* pSessionContext,
                                                uint32_t nCommandID,
                                                uint32_t nParamTypes,
                                                TEE_Param pParams[4])
{
    switch(nCommandID)
    {
```

```
    case CMD_SAMPLE_ROT13:
        if (nParamTypes != TEE_PARAM_TYPES(
                                TEE_PARAM_TYPE_MEMREF_INOUT,
                                TEE_PARAM_TYPE_NONE,
                                TEE_PARAM_TYPE_NONE,
                                TEE_PARAM_TYPE_NONE))
        {
            tlDbgPrintf(TATAG "bad params\n");
            return TEE_ERROR_BAD_PARAMETERS;
        }
        uint8_t* pOutput     = pParams[0].memref.buffer;
        uint32_t nOutputSize = (uint32_t)pParams[0].memref.size;
        for(i=0; i<nOutputSize; i++) {
            pOutput[i] = pOutput[i]+13;
        }
        return TEE_SUCCESS;
    }
}
```

The code shows the selection of a command with the switch statement and the treatment of the ROT13 command. The first check verifies the parameter type: Here only one parameter of type memory reference and direction input and output is accepted. The variables are declared to have easier access to the provided parameters. A simplistic loop applies the rot13 disguise to the data.

In addition to OUT parameters, entry points also have a return value that can be used to communicate the overall result of the operation.

The Trusted Application developer defines the service interface by choosing the required command IDs and by associating a set of typed parameters for each command. Those definitions, together with application-specific data types and return codes should be put into a header file.

The following header illustrates a sample service interface:

```
/** Command ID's */
#define CMD_SAMPLE_ROT13    1
#define CMD_SAMPLE_HELLO    2

#define CMD_ROT13_PTYPES TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_INOUT,
TEE_PARAM_TYPE_NONE, TEE_PARAM_TYPE_NONE, TEE_PARAM_TYPE_NONE)
/**
 * ROT13 parameter description.
 * The ROT13 command will apply disguise to data provided in parameter 1.
 * Only characters (a-z, A-Z) will be processed.
 * Other data is ignored and stays untouched.
 */

#define RET_ERR_INVALID_BUFFER   1
#define RET_ERR_INVALID_COMMAND  2
```

3.3.3 Extended APIs

Some extended APIs such as the Trusted User Interface API, the DRM API and others are also available for Trusted Applications developed with the GlobalPlatform APIs. The exact names and function signatures for these APIs are indicated in the `<t-base API Documentation`.

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.

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.

For the debugging on a device or <t-base emulation, a development device with <t-base is needed.

Hint: Commercial devices [ComDev] with <t-base can be used to run Trusted Applications. But debugging interfaces like COM or JTAG are disabled, user rights are restricted and older release software prevents debug outputs.

4.1.2 DS-5™ / ARM Compiler

The ARM Development Studio 5 (DS-5™) is a development environment including compiler and debugger for ARM based 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. To build Trusted Applications with the ARM Compiler, the Professional Edition is required.

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

- install ARM-DS5 compiler toolchain 5 from https://silver.arm.com/browse/DS500/
  Use of version 5.10 or newer is recommended.
- obtain ARM-DS5 license from ARM or internal lic-server
  There is a 30 days eval license that requires the MAC-address of the developer PC
  The node lock license is recommended when bandwidth plays a role.

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 GCC / Linaro Compiler

The Linaro GCC 4.7.3 is GNU development tool-chain for ARM Embedded Processors. It includes compiler tools and debugger for ARM base platforms. The Linaro Compiler can be obtained from this web site: https://launchpad.net/gcc-arm-embedded/4.7/4.7-2012-q4-major.

You could install the whole tool-chain by simply untarring the archive inside the chosen directory.

1. Download the archive
   On Linux the archive name is gcc-arm-none-eabi-4_7-2012q4-20121208-linux.tar.bz2
2. Untar the archive in the current folder

```
tar xjf gcc-arm-none-eabi-4_7-2012q4-20121208-linux.tar.bz2
```

4.1.4 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 r8 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).

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 CA together with NDK may not work.

4.1.4.1 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"
```

4.2 QUICK STARTING GUIDE

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

Please read also the chapters:

- Compiling and Testing Trusted Applications
- Compiling and Testing Client Applications

They hold more detailed information and might answer your questions.

4.2.1 File structure

The <t-sdk package provides everything you need for TA and CA 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 TAs and CAs.

‹ \

- 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, CAs 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, CA and Android app using the legacy APIs.

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 Client Application (CA)

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

  ```
  export COMP_PATH_<yourTAname>=
  ${COMP_PATH_ROOT}/../projects/<yourProject>/<yourTAdir>/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 CA and TA.

4.2.4.1 On a Commercial Device (running as 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.

See 4.3.3 for more details on how to use SP-PA and provisioning of SP-TA.

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`)

```
TA_UUID := 06010000000000000000000000000000
TA_SERVICE_TYPE := 3 # System Trusted Application
TA_KEYFILE := <your path>/pairVendorTltSig.pem
```

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

```
adb push 06010000000000000000000000000000.tlbin data/app/mcRegistry/
adb push <myCAname> data/app/
```

4. start your CA

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

Use a SP Trusted Application

See 4.3.3 for more details on how to use SP-PA and provisioning of SP-TA.

4.2.5 Creating a secure application following Global Platform scheme

You can use the Rot13-GP <t-sdk -Sample to create your own Trusted Application and Client Application using the Global Platform APIs.

4.2.5.1 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 tee_internal_api.h (TEE_LogvPrintf TEE_LogPrintf TEE_DbgPrintf TEE_DbgvPrintf).
- call the build script

```
./build.sh
```

- use the same build options as for TAs following the legacy API.

4.2.5.2 Client Application (CA)

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

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

- use the same build options as for CAs following the legacy API.

4.2.6 Running a Global Platform application

You can also run secure applications following the Global Platform scheme. Secure applications following the Global Platform scheme can be either System-TAs or SP-TAs.

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

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.

Use a signed GP System Trusted Application:

1. Use the RSA System TA sample private key that comes with the <t-sdk
   `(t-sdk-rXXX\Samples\Rot13_GP\ TASampleRot13\`
   `Locals\Build\pairVendorTltSig.pem)`

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

```
GP_ENTRYPOINTS := Y
TA_UUID := 08010000000000000000000000000000
TA_SERVICE_TYPE := 3 # System Trusted Application
TA_KEYFILE := <your path>/pairVendorTltSig.pem
```

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

```
adb push 08010000000000000000000000000000.tabin data/app/mcRegistry/
adb push <myCAname> data/app/
```

4. start your CA

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

Use a GP SP Trusted Application

Global Platform defines a UUID scheme for Trusted Applications that is based on a key pair per TA. The hash of the public key is used as UUID of the TA. The private key is used to bind the UUID to the TA binary. This prevents UUID impersonations.

1. Use the symmetric SP TA sample private key that comes with the <t-sdk
   `(t-sdk-rXXX\Samples\Rot13_GP\ TASampleRot13\`
   `Locals/Build/keySpTl.xml)`

2. Use the RSA UUID sample private key that comes with the <t-sdk
   `(t-sdk-rXXX\Samples\Rot13_GP\ TASampleRot13\`
   `Locals\Build\pairUUIDKeyFile.pem)`

3. Compile your Trusted Application as a SP TA (adapt `makefile.mk`)

```
GP_ENTRYPOINTS := Y
TA_UUID := d51a83c9474a5655af2a58dcfd2b1d37
```

```
TA_SERVICE_TYPE := 2 # Service Provider Application
TA_KEYFILE := <your path>/keySpTl.xml
TA_UUIDKEYFILE := <your path>/pairUUIDKeyFile.pem
```

4. load both modules (from TA/CA out-folders) onto your device

```
adb push d51a83c9474a5655af2a58dcfd2b1d37.tabin data/app/mcRegistry/
adb push d51a83c9474a5655af2a58dcfd2b1d37.spid data/app/mcRegistry/
adb push <myCAname> data/app/
```

5. start your CA

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

4.3 COMPILING AND TESTING TRUSTED APPLICATIONS

The following section describes what you need to get started writing a 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 once 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 3.2.1, Trusted Application and Client Applications, inter world communication is done using sessions and invoking commands with memory references. See the next section for a basic example of these IPC mechanisms.

4.3.1 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

4.3.2 Compiling and signing a Trusted Application

Trusted Applications are being 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. [4.1.3]

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 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. 4 <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 productive 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 lets you compile and sign a Trusted Application by 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).

Nota:

The TA_KEYFILE parameter of makefile shall not be changed as Trusted Application containers are preinstalled.

4.3.3 Using the SPPA library for provisioning TAs

SPPA is an embedded component within the rich OS application which implements the service provided by a Service Provider.

It is a library which is coupled with each Service Provider Application (SPApp) using <t-base client API and which needs <t-base life cycle management.

The SP-PA component is dedicated to the containing application. It performs TA related life cycle management and communicates with the Trusted Application Manager (TAM) server in the <t-base eco-system. There are as many SP-PA instances on the device as <t-base client applications. The SP-PA is provided by and bounded to the TAM.

The whole <t-base ecosystem including the interaction with the OTA components is presented by the diagram below:

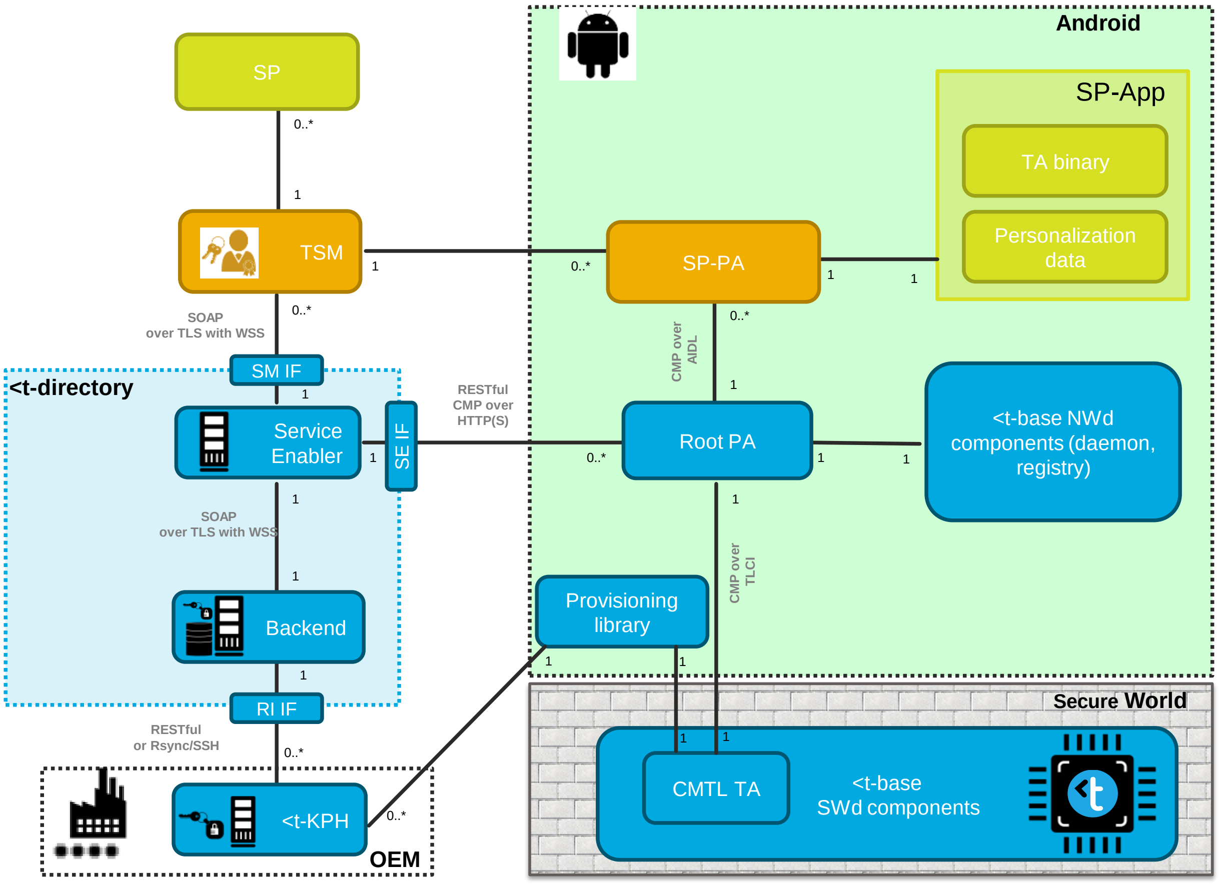


Fig. 2. <t-base ecosystem overview

4.3.3.1 Deployment configuration

Trusted Application binaries may either be bundled with the rich OS application (APK) or come from the TAM. The TAM may transfer TA binaries to SP-PA at installation time. SP-PA in turn shall call the SP-App via the *SpAppCbService* to save the TA binaries in the SP-App user space.

Option 1: TA bundled within the rich OS application: The TA binary contained in the APK is encrypted. The key is specific for the TA, but common for all devices. So the same key shall be used to encrypt the TA binary on all devices. The TAM shall put this TA key into the TA container. This solution is less secure than the option 2 because TA binary encryption is not device specific. This option is used for testing SP-TA provisioning with Trustonic Test Server.

Option 2: **TA binary coming from the TAM:** The TA binaries are transferred from the TAM, the server can create a specific key for each TA and each device. In this operation mode the TAM shall create new TA keys, create the TA containers on the device and put the keys into these TA containers using CMP. Next the TAM shall encrypt the TA binaries, which are stored on the TAM, with the newly created keys, before transferring them via SP-PA into the user space of the SP-App. The Option 2 is available only for customers who work with a TAM company.

4.3.3.2 Provisioning workflow

The SP-App is downloaded from the application store (for instance Google Play). When started the first time the SP-App logic calls the initialization function offered by the SP-PA API. The SP-PA shall drive the rollout process further on. In detail the process shall be as follows:

1. All starts when the end user downloads a <t-base client application to the device and starts installation. The download comprises the service provider application, the TA binaries, the TA connector (TLC) and the linked SP-PA library
2. When the application is started up it shall realise that it requires an Over The Air provisioning of its TAs. Thus in the initialisation phase SP-App calls the embedded SP-PA.
3. The SP-PA will communicate with the root PA in order to request the root and SP containers (if not present already).
4. The root PA will return immediately, if root and SP containers are already in place. Otherwise it will contact <t-directory Service Enabler (SE) and request the commands to perform root registration (if necessary) and the creation of the SP-container.
5. The SE will will generate and store the SP-container key internally and hold the key available for pickup by the TAM.
6. After all root operations are completed SP-PA connects to the TAM.
7. The TAM shall retrieve the SP-Container key from SE. So this way the TAM gets access to the SP-Container. The TAM exchanges the container key and activates the container on the device.
8. The TAM generates the commands to register and activate the TA container on device.

   If personalization of the TA is needed the step will be split into two steps and an additional command will be created for each personalization data to send to the device. After personalization commands were processed the TA container is activated.
9. For deployment option 2 only: Before closing the exchange, if the Trustlet Application Binary needs to be send to the device, the TAM will push the appropriate content to the device within a XML envelope. The SP-App will store the binary at a correct location depending of the platform (see next section).
10. If done SP-PA returns and the application is ready for operation.

4.3.3.3 Integration from a Client Application

The TSdkSample from the <t-sdk shows an example of the SPPA library implementation.

Here is the path to follow in order to get SPPA libraries functions:

1. Copy SPPA jar file to your SP Application project libs/ folder:

```
cp -v ${COMP_PATH_OTA}/sppa-*.jar libs/
```

2. Import the SPPA library in your Java code:

```
import com.trustonic.tbase.ota.sppa.ifc.Utils;
import com.trustonic.tbase.ota.sppa.ifc.SpPa;
import com.trustonic.tbase.ota.sppa.ifc.SpPaService.ProgressCb;
import com.trustonic.tbase.ota.sppa.ifc.SpPaService.ProgressState;
import com.trustonic.tbase.ota.sppa.ifc.IfcConstants;
     import com.trustonic.tbase.ota.sppa.ifc.SpAppCbService;
```

3. Create a provision object and call the install() method:

```
        // Create Service Provider Provisioning Agent (SpPa) instance.
        mProvision = new Provision(activity, mPrefs);
        if(mProvision != null) {
            // Install TA.
            mProvision.install();
        }
```

The Provision.java and Service.java files can be used as a starting point for using the SPPA library.

The provision class constructor acts as follows:

- Read information for SM and SE servers URL to use from the application resources file file
- Instantiate a ProvisionApp callback
- Instantiate a SPPA object
- Instantiate a Service object linked to the SPPA object
- Instantiate a Provisioning Callback
- Connect to SPPA
- Downloads Root and AuthToken containers if not present on the device

Then the Provisioning install() method acts as follows:

- Waits for SPPA connection to be established
- Launches Service install() method with parameters:
  - Provisioning callback
  - Package(SP App) name
  - ProvisionApp callback
- Launches the SPPA installTA() method with parameters:

- Service callback
- Package(SP App) name
- ProvisionApp callback

The installTA() method provisions the containers for the TAs needed by the current package in the form <UUID>.<SPID>.tlcont.

The SP container is also updated with the UUIDs of the TAs installed.

The TAs just provisioned are now ready to use.

4.3.4 Running a Trusted Application in a Development Environment

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

4.3.5 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 SWd 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.

4.3.5.1 Logging

Trusted Application can use printf-like logging. The log is written to the Linux kernel log and can be retrieved via `adb shell dmesg` or using the serial port of your Arndale board. The function `tlApiLogPrintf(…)` writes a formatted string to the Linux log. The helper function `tlDbgPrintf(…)` can be used for logs that are only included in the TA binary in debug builds.

Apart from that, TAs can return error codes and error strings in the parameters of a command.

4.3.5.2 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.

4.3.5.3 Debug Agent

4.3.5.3.1 DS-5 setup

4.3.5.3.1.1 Install DS-5 Version 5.17.

- It is available at the ARM download page: DS-5 download, this requires an ARM account
- Here are some commands to download and setup the DS-5 debugger.
- Download

```
wget http://ds.arm.com/media/downloads/DS500-BN-00019-r5p0-17rel0.tgz
```

- Setup dependencies

```
sudo apt-get install ia32-libs
sudo apt-get install libwebkitgtk-1.0-0
```

- Extract and install

```
tar xvzf DS500-BN-00019-r5p0-17rel0.tgz
sudo ./install_x86_64.sh
You should accept the license and confirm the installation folder
/usr/local/DS-5
```

4.3.5.3.1.2 Obtain a license

- DS-5 comes with 30 days evaluation license.
- There are 2 versions of the evaluation license. The "community license" is inadequate for debugging purpose only the "full license" allows to debug Trusted Application.

4.3.5.3.2 Install debug agent

- Launch DS-5

```
/usr/local/DS-5/bin/eclipse
```

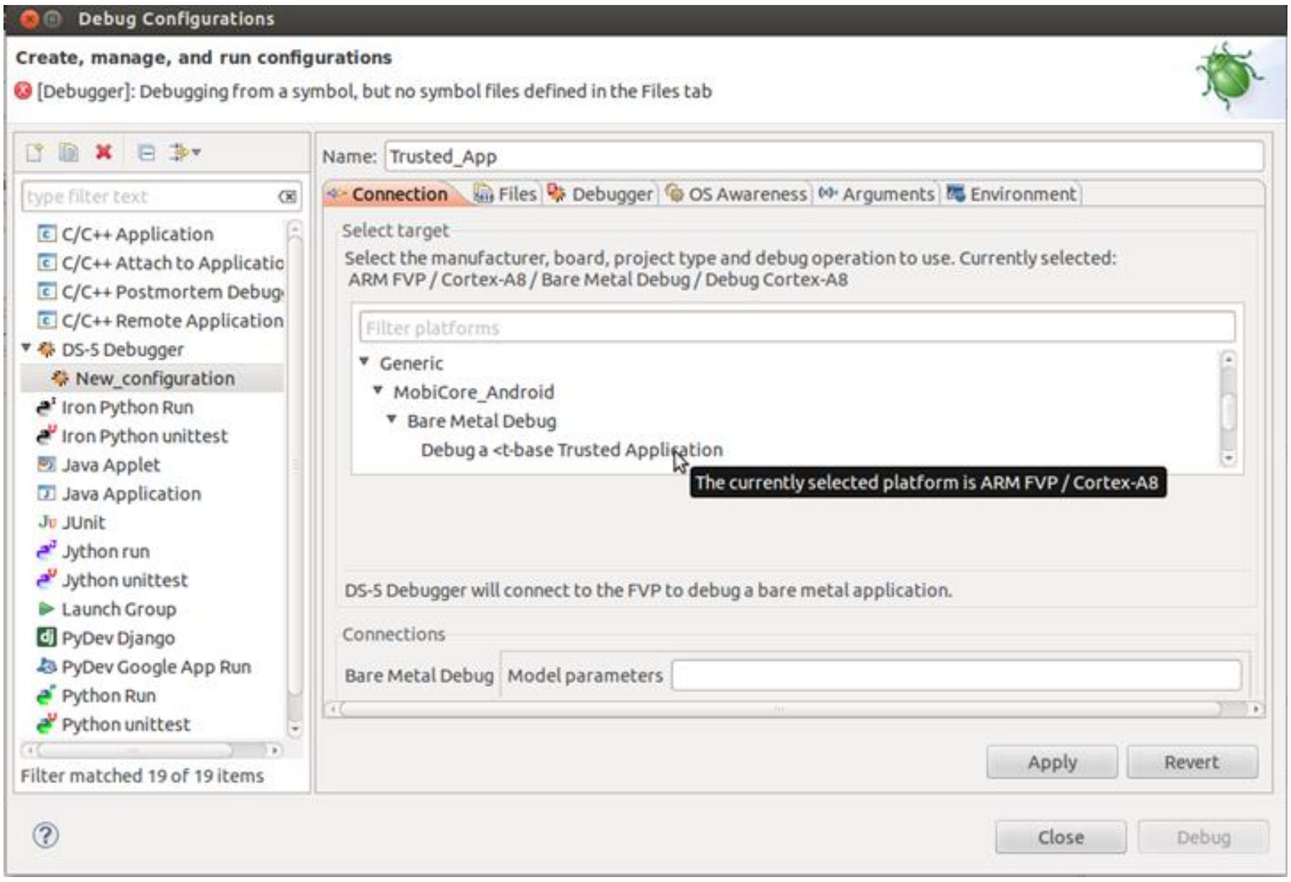
- Go to '`Window->Preferences->DS-5->Target Database`'. (See '`Window->Preferences->DS-5->Configuration Database`' on DS-5 v5.12 and onwards)
- Click '`Add...`', choose the directory into your t-sdk: `t-sdk-rX/Tools/DebugExtension`
- Click '`Apply`', then click "`Rebuild database`"
- Restart DS-5

4.3.5.3.2.1 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 named. 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 the platform:
  `Generic >> MobiCore_Android >> Bare Metal Debug >> Debug a <t-base Trusted Application.`


Debug Configurations

Create, manage, and run configurations

[Debugger]: Debugging from a symbol, but no symbol files defined in the Files tab

type filter text

- C/C++ Application
- C/C++ Attach to Applicatio
- C/C++ Postmortem Debug
- C/C++ Remote Application
- DS-5 Debugger
  - New_configuration
- 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

Filter matched 19 of 19 items

Name: Trusted_App

Connection | Files | Debugger | OS Awareness | Arguments | Environment

Select target

Select the manufacturer, board, project type and debug operation to use. Currently selected:
ARM FVP / Cortex-A8 / Bare Metal Debug / Debug Cortex-A8

Filter platforms

- Generic
  - MobiCore_Android
    - Bare Metal Debug
      - Debug a <t-base Trusted Application

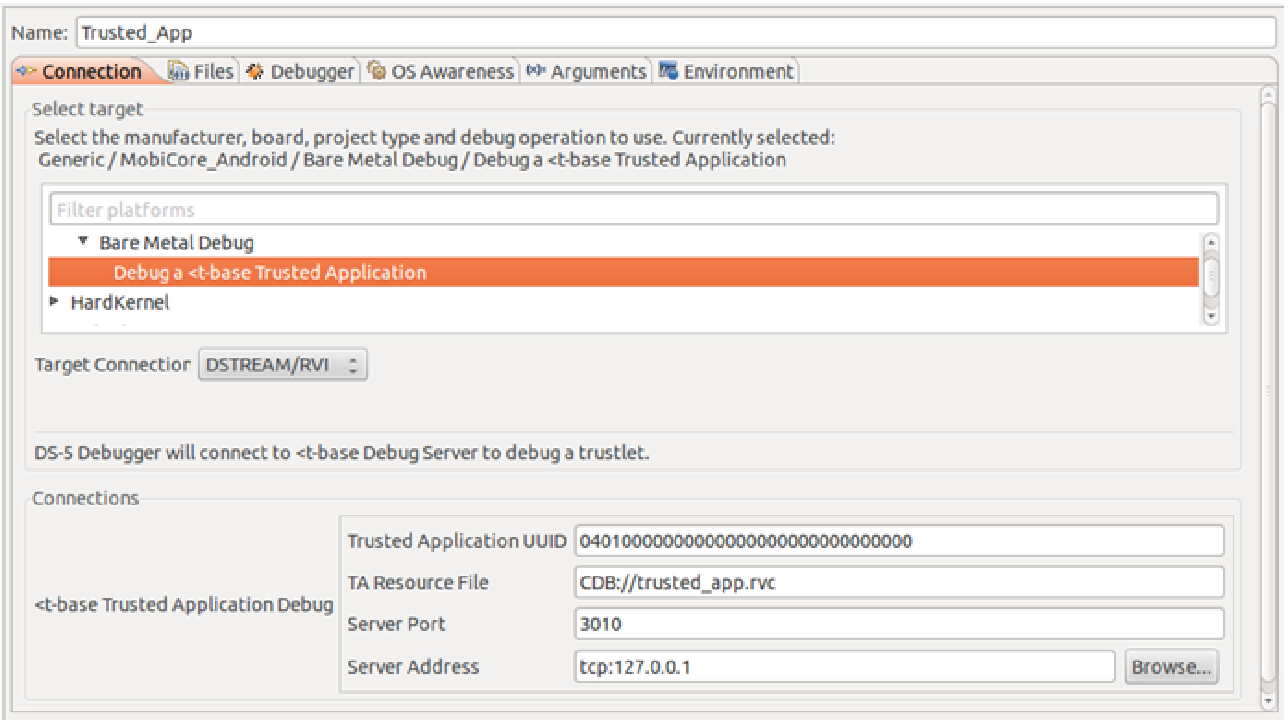
The currently selected platform is ARM FVP / Cortex-A8

DS-5 Debugger will connect to the FVP to debug a bare metal application.

Connections

Bare Metal Debug | Model parameters

- Keep the default parameters, as most of them are currently ignored, and just manually add the IP address of your target.


Name: Trusted_App

Select target

Select the manufacturer, board, project type and debug operation to use. Currently selected:
Generic / MobiCore_Android / Bare Metal Debug / Debug a <t-base Trusted Application

Target Connection: DSTREAM/RVI

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

NB: 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`

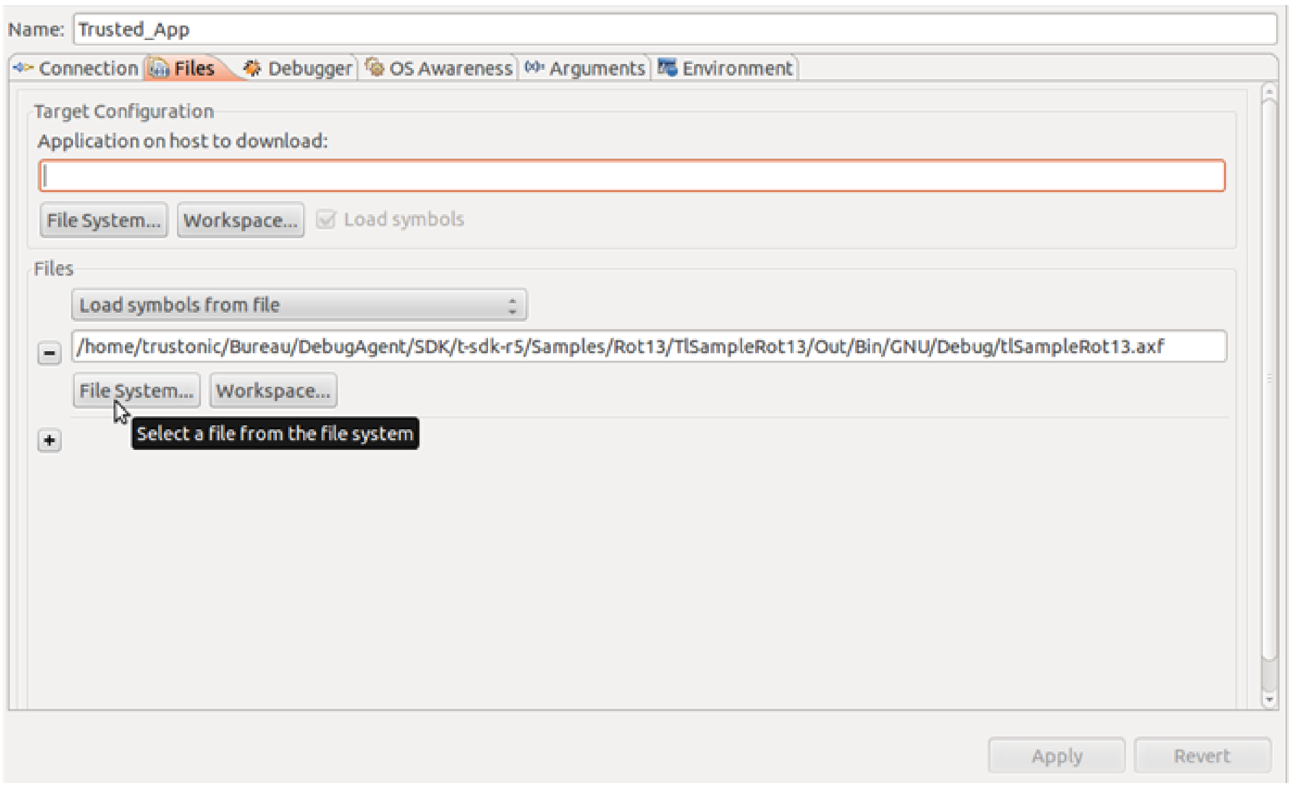
In this case you should use the IP address 127.0.0.1 and port 3010.

- 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 modify '`Boards\Generic\MobiCore_Android\trusted_app.rvc`' and replace all occurrences of '`04010000000000000000000000000000`' to UUID of your Trusted Application
  NB: Because this is not convenient, there is a script inside your t-sdk directory to help:
  `t-sdk-rX/Tools/prepare_debug.sh`

  This script performs the adb forwarding and could also set the UUID.

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, once I have built the tlSampleRot13 Trusted Application inside your t-sdk package, you could pick the .axf binary as shown in the screenshot below.


Name: Trusted_App

Target Configuration

Application on host to download:

Files

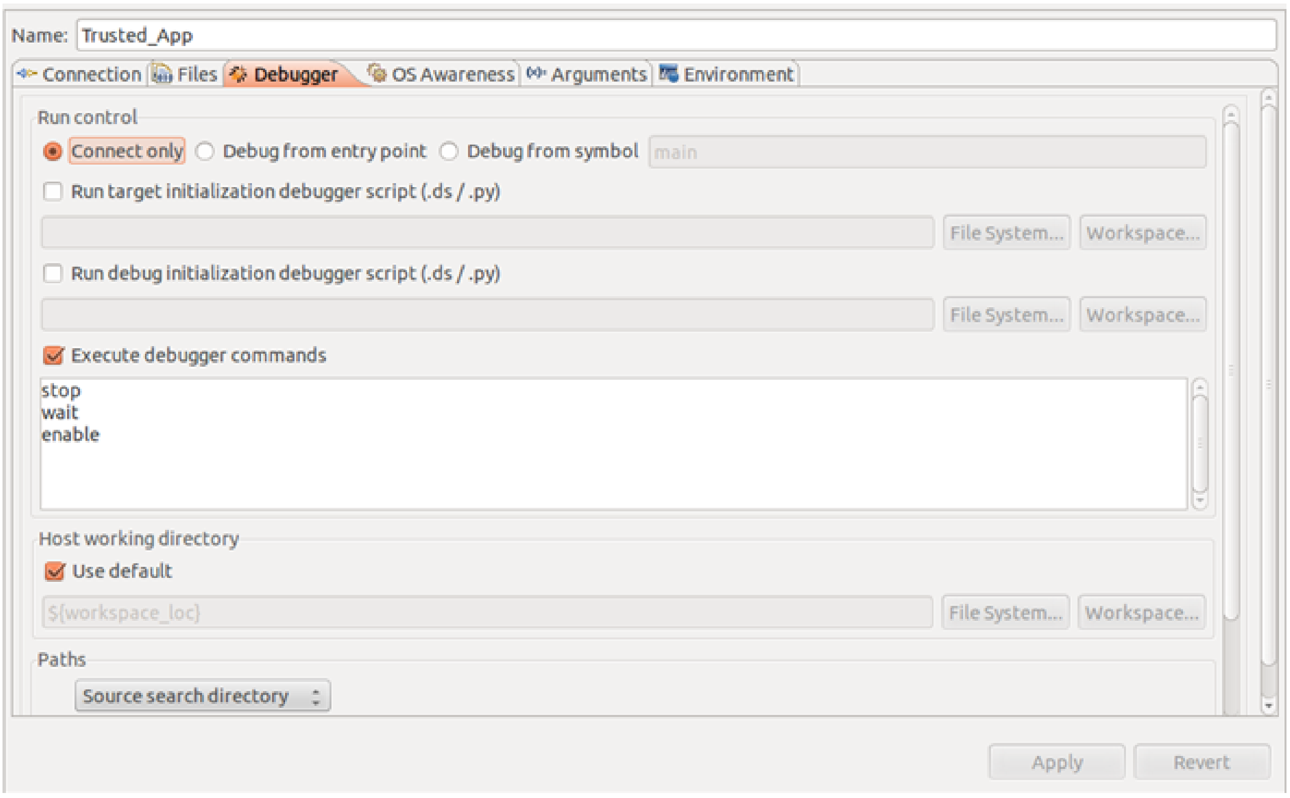
Load symbols from file

/home/trustonic/Bureau/DebugAgent/SDK/t-sdk-r5/Samples/Rot13/TlSampleRot13/Out/Bin/GNU/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
```


Name: Trusted_App

Connection | Files | Debugger | OS Awareness | Arguments | Environment

Run control

[✓ Connect only] ○ Debug from entry point ○ Debug from symbol main

☐ Run target initialization debugger script (.ds / .py)

☐ Run debug initialization debugger script (.ds / .py)

☑ Execute debugger commands

```
stop
wait
enable
```

Host working directory

☑ Use default

${workspace_loc}

Paths

Source search directory

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

Arguments tab

- Use default. It should be empty

Envirnonment tab

- Use default. It should be empty.

Event Viewer tab

- Use default. It should be empty

4.3.5.3.2.2 Debug Agent setup on device

- The tldebug.apk is already installed on your Arndale board (flashed with our t-sdk).
- The Debug Agent driver (`'07030000000000000000000000000000.tlbin'`) should be on the device inside the `/data/app/mcRegistry` directory.
- Make sure that `libMcTlDbg.so` is available on device (e.g. under /sytem/lib/).
- However you could re-install the tldebug if needed.
  Inside the t-sdk package:

```
adb install ./Tools/DebugAgent/tldebug.apk
adb remount
adb push ./Tools/DebugAgent/libMcTlDbg.so /system/lib/
adb        push        ./Tools/DebugAgent/07030000000000000000000000000000.tlbin
/data/app/mcRegistry/
```

- Launch `com.rtec.tldebug` as below

```
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
- In the Debug Configurations, start the debug procedure by pressing the Debug button. Now you should get a window indicting that the debugger tries 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".

4.3.5.3.3 Debugging 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.

```
TA_FLAGS := 4
```

- Go to DS-5 '`Debug Control`' view. Select target and then '`Connect to Target`'
- You should see that the tldebug has entered a wait loop, waiting for the user to run a Trusted Application Connector which will load the Trusted Applciation.

```
$ adb logcat
...
I/tldebug ( 2715): GetCapabilitiesSize - called, rvm_dev = 1
I/tldebug ( 2715): GetCapabilities - called, rvm_dev = 1
I/tldebug ( 2715): GetCapabilitiesSize - called, rvm_dev = 1
```

- From board, run you TLC which will load your Trusted Application.
- Once DS-5 connection succeeds, DS-5 will break at the beginning of the TA and then you can debug your Trusted Application.

4.3.5.3.4 Error Cases

- If the initial connection to the TA fails, make sure permissions for /dev/mobicore are set to 777 as the app does not run as root

4.4 COMPILING AND TESTING CLIENT APPLICATIONS

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

Building a CA 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 suitable to compile and build executable, shared- and static libraries.

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

4.4.1 Client Application Structure

The Client Application 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 CA is composed of the following components:

- One or more generic CA methods using the TEE Client API to communicate with the Trusted Application in the SWd.
- A shared library written in C/C++ wrapping the generic CA 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.

4.4.2 Compile a CA with the Android NDK

The Android NDK is used to build the NWd’s native code, namely the CA consisting of the shared library and an (optional) binary for testing purposes.

4.4.2.1 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.

4.4.3 Run your Client Application

4.4.3.1 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
```

4.4.3.2 Connect to the Device via Ethernet

Depending on your device, they 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. 5 ADB commands

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

4.4.3.3 Upload and Test

In order to test your CA 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 CA 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 CA:

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

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

4.4.3.4 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.

4.4.3.5 Client Application 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 CA'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.

4.4.4 Debug

This section should give 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. 6 Linux system debugging commands

4.4.4.1 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.

4.4.4.2 GDB

For debugging your C/C++ CA 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.

4.4.4.2.1 Debug stand-alone binaries

This section describes how to debug a stand-alone CA 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.

5 MOBICONVERT MANUAL

MobiConvert is Trustonic's file conversion tool that creates various binary objects required for deployment of secure services. Its main purpose is to take a service executable image (ELF file of a Trusted Application or driver) as <in-file> and to create a <t-base loadable service image as <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.

Other modes of operation are the creation of a UUID, of a RSA key pair, to generate a UUID from a RSA key and to display file information of converted files.

5.1 RSA KEY GENERATION

To encrypt Trusted Applications of the Service Provider type, a RSA key is required. You can generate a RSA key pair using MobiConvert:

```
java -jar MobiConvert.jar --genkey
```

Example:

```
The generated RSA key is:
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAq7vDwXYn5MElF5Ny+JeOH2DFY/JAcDNwG3xXT8n7jgmWg/Sc
ctYM/xu6x/QtkCKUquaV9BiRW8olZyPeMZ83+K1lpFXZl12gCHejItKgyifVx03h
XsBY+G2rGGQH64kk2dkjtfnzRxUkTd1g5flMEjnd04HLi+Sm6iXjfssr1YkAl9TQ
+J0QZQ7T18RzCs7w2Pfu4QIrU2zFfeS1AG/en3U+Iiu278i54U4dZn3zJplSFX0K
WIQiozCU0ZFv+xBm+xvzv3+kOpDgFmjeTvcV8nl7Ywo6+Mu55IvsAAwQkFX2rMv4
rGyURaFtRS5gdHdCWSR2s42Ozi05WwYzlbk4mQIDAQABAoIBAQCoqDfBSPQ3AcUW
2VWBdP48LMLOvHyydKH2LHBnSPvHa/0pTNNVKIkNBcOzWkhhFMUi7mB4oKQpHGcN
Wzl9TFwg2tJGyZVvxaBEkJJjwa3tu5+FJCRg9NCr8rCkvKDWnhLV7B3ZO0fEGKxV
2EOwt9wQzknfzcoEcqPGsz0wKgK7qzuh8/6KmCfjLBgbxvkYZB+3NYLyNibzTx47
fhaCVqwdgbnUy4YQ2PIEEO+uNYIqbV7QQBWEClFrHd8/5JNEZAw4XUzXA0XOBqN0
P5jWZntfAqFfr1aQAOHs5fiGi2a4w40cg5+sJBeCrt1rWcpW8Zmm2KU6F0bNM1Q5
xhQ4Sx2xAoGBAN49lQlY2mhOW2bGZqjZ5xug6G4k/AJoQNdvFYaVMEMKImC94Pp6
jMVv5Z+abhVuPAoKmkRvChW02QKTI6DbK49x3vAwlnN7bEwo8L4PuW8OtRm08S+U
rza6iTujsnMVfEh9j/P/6ewfFJyXCOyMQxlNmQQkaIWk8d5mv1xZV2T1AoGBAMXS
FDeF5XMHP6RCYk8GL+A3HO8fGQUs7zAyPzzdAatTRmLgZw9gzldTe2rEuhE2XhiI
k8xHhymNC3Yj5GRQhVZRwD+tDzgt5qGbQmLsw2sKCb0XoundJagiofzZTj0WAVT+
fvY5iheRSofcRJLQFLA+lgXxu0KZ8rGXJMxBs96VAoGBAMumlObe1C1W+Gzii/pY
y23G8pbUL1apYBnKgmg0V+hm5f/On9YH7O2Tz1CE/DGJNV1iP+FL+2rOsTmpybFC
hdVJ3Kgvbf7e7+uObKVN1XgOeyfWZllan4DASLctF35cBuqKnRpTvXERPhsMUDIr
ieUq9XgVQO6OqtFJSDwA5pPtAoGBALAGO08cqgstDAhRucCvtLJC2FA+z7i3Py8X
xwWVcwLMWvlozMv2TCWQd2WOIDNouVoDTeCcVT038FbzoStSKxOgMv12NPC8h1iO
GwiDvW/lwryr559J1VRDXPjtNJ1Ok2jZ/IeEs8g81KEH80zgM0iQqFYpv4OIEVjN
MUU/wZnxAoGAHSnx3KM8xlD6orUkPqsxJWh9uUfW4VPYqxspJe1RRaAxTql2SJLX
kaBL094pUB5XLyVsc1eJe8Hb+E+VPdsBmTZ+SAP49L7Ax9YS3x1T/QhcrrstREOW
zxSMu4peXV7AewV6RFD/1uX9Ga3ftr8bm199mN4urrHJnIMtw55wM+I=
-----END RSA PRIVATE KEY-----
```

You can copy this text into a file at Locals/Build named pairVendorTltSig.pem.

5.2 UUID GENERATION

5.2.1 For legacy services and drivers

To generate a RFC 4122 UUID for your new legacy service, use the following command-line:

```
java -jar MobiConvert.jar --genuuid4
```

Example:

```
The generated uuid (type 4) is
ea846ca0bf9c425ebbe7b4600f5a3189
```

You can copy this UUID into your services Makefile as TA_UUID.

5.2.2 For Global Platform Trusted Applications

For Global Platform TAs, the UUID is derived from a RSA key pair. Generate a key pair using MobiConvert and store the key under Locals/Build/pairUUIDKeyFile.pem.

Then call MobiConvert again to derive the UUID from this key file:

```
java -jar MobiConvert.jar --printuuid --uuidkeyfile Locals/Build/pairUUIDKeyFile.pem
```

Example:

```
java -jar MobiConvert.jar --printuuid --uuidkeyfile uuid.pem
cb89f66f-f771-5614-a2b1-3f0bc3229e83
```

Remove the slashes and copy this UUID into your services Makefile as TA_UUID.

Note, you need to specify the UUID key file also when converting your Trusted Application.

5.3 CONVERSION FEATURES

Detailed description of relevant options:

```
-b, --bin <in-file> The path to the ELF input file (xxx.axf).
-s,--servicetype <integer> Service type of input file
       1: Driver
       2: Trusted Application
       3: System Trusted Application
-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
-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
-gl, --gp_level <string> Set string to ‘GP’ to create a TA following GP scheme.
-uk, --uuidkeyfile <string> for GP TAs, the path to an RSA key pair file to derive the
UUID from and to embed the UUID-attestation into the TA binary.

-iv, --interfaceversion <major.minor> The interface version to be used
       Mandatory for service type drivers.
       The range of major and minor shall be each between 0-65535.
-m, --memtype, -m <type> The type of memory that should be used for the service.
       0: Internal memory preferred. If available, use internal memory, otherwise use
external memory.
       1: Use internal memory,
       2: Use external memory (Default).
-i, -numberofinstances <num> Max. number of concurrently active instances (default = 1)
       1: for drivers there can be only one instance
       1-16: for service type 2 and 3
```

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)

Options to create Trustonic test binaries:

```
--relaxedout, -rO   Relaxed naming rules for output image filename
                    (you should however keep a name beginning with the uuid).
--relaxedlen, -rL   No size limit checks for executable
--nosign            Convert a file and create the header, but do not add the signature.
```

5.3.1 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 1 -d
<driverId> -iv <major.minor> [-i <numberOfInstances>] [-n <numberOfThreads>] [-m
<memoryType>] [-f <flags>]

-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.
-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
```

5.3.2 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 2 [-m <memoryType>] [-i <numberOfInstances>] [-n <numberOfThreads>] [-f <flags>]
```

5.3.3 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 3
       [-f <flags>] [-m <memoryType>] [-i <numberOfInstances>] [-n <numberOfThreads>]
```

5.3.4 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>
```

5.3.5 Examples

Converting a Driver:

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

Converting a Service Provider Trusted Application:

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

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
```

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
```

Converting a System Trusted Application:

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

Example of using the Header mode:

```
java -jar MobiConvert.jar -h 04010000000000000000000000000000.tlbin
MODE = Show trustlet/driver header only

###### Trustlet header: ######
Magic                             = MCLF
Version                           = 2.3
Flags                             = Debuggable
MemType                           = external
Service type                      = Service Provider Trustlet
Service Type                      = Service Provider Trustlet
NumberOfInstances                 = 16
UUID                              = 04010000000000000000000000000000
DriverID                          = 0
NumberOfThreads                   = 1
CodeAddr                          = 0x1000
CodeLength                        = 3184
DataAddr                          = 0x3000
DataLength                        = 0
BssLength                         = 6328
Entry Addr.                       = 0x1B29
Interface Version                 = 0.0
permittedSuid                     = ffffffffffffffff0000000000000000
GPLevel                           = 0
attestationOffset                 = 0
```

5.4 HELP OUTPUT

```
java -jar MobiConvert.jar -?
MobiConvert V1.2
  Copyright (c) 2013-2014 TRUSTONIC LIMITED. All rights reserved.

usage: [-b <pathToBinary>] [-o <pathToOutput>] [-k <pathToKeyFile>] [-s
           <serviceType>|-?|-h <pathToFileToRead>|-g] [-f <flags>] [-m
           <memoryType>][-i <numberOfInstances>] [-n <numberOfThreads>]
           [-d <driverId>] [-iv <major.minor>]
 -?,--help <arg>                        no argument: all parameters, 1:
                                        driver parameters, 2: Service
                                        Provider Trustlet parameters, 3:
                                        system Trustlet parameters, 4:
                                        header mode parameters
 -b,--bin <file>                        name of the ELF input file
 -d,--driverid <integer>                driver identifier
 -f,--flags <integer>                   0 = no flag,
                                        1 = permanent,
                                        2 = service has no WSM control
                                        interface,
                                        4 = debuggable,
                                        add the value of the different
                                        flags you want
 -g,--genuuid4                          generate a version 4 UUID
                                        according to RFC 4122 and exit
 -gk,--genkey                           generate 2048-bits RSA key pair
 -gl,--gp_level <string>                GP level flag (GP = GP API is
                                        required)
 -h,--header <arg>                      show the header of the file given
                                        as argument
 -i,--numberofinstances <integer>       number of instances (1 for
                                        drivers, between 1 and 16 both
                                        included for Trustlets)
 -iv,--interfaceversion <major.minor>   give the version of the interface
                                        in the format major.minor
 -k,--keyfile <file>                    name of the key file
                                        [path]filename.xml, which contains
                                        key (trustlet) or key pair (driver
                                        or system trustlet)
 -m,--memtype <integer>                 0 = internal memory prefered, 1 =
                                        internal memory used, 2 = external
                                        memory used
 -n,--numberofthreads <integer>         number of threads (1 for
                                        Trustlets, between 1 and 8 both
                                        included for drivers)
    --nosign                            Output is unsigned.
 -o,--output <file>                     Output file name. Name must be
                                        [path]UUID.extension (UUID 32
                                        char. length in hex format) The
                                        extension is tlbin for trustlets
                                        and drbin for drivers. Extension
                                        .raw is appeneded for unsigned
                                        output file.
 -pu,--printuuid                        Print UUID generated from RSA key
                                        to standard output
 -rL,--relaxedlen                       no Limit on the trustlet length
 -rO,--relaxedout                       use less checks for the output
                                        file name
 -s,--servicetype <integer>             service type of input file:
                                        1: driver
                                        2: trustlet
                                        3: system trustlet
 -uk,--uuidkeyfile <string>             RSA key pair that proves UUID
                                        ownership
```