Trusted Foundations™ for Nvidia Tegra3

Product Reference Manual

Date of Issue:	August 6th, 2012
Reference & Version:	CP-2011-RT-513-V1.7
Classification:	Confidential
Number of pages:	54 (including 2 header pages)

PREFACE

Version History

Version	Date	Status	Modifications
1.0	October 10th, 2011	Issued	First issued version
1.1	November 18th, 2011	Issued	Document updated for Neon support and minor clarifications
1.2	January 16th, 2012	Issued	Document updated to add postlinker Multi-stage support description
1.3	February 03rd, 2012	Issued	Document updated to add implementation notes on dynamic boot arguments
1.4	March 09th, 2012	Issued	Document updated to complete implementation notes on dynamic boot arguments (trace driver added)
1.5	May 05th, 2012	Issued	Document updated to add sections on memory profiling tool.
1.6	June 04th, 2012	Issued	Document updated to add note on TF client API in kernel
1.7	August 6th, 2012	Issued	Document updated to add details on tasks profiling tool.

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual describes how to port and integrate the Trusted Foundations™, on the Tegra3 platform.

About the Trusted Foundations™ for Tegra3

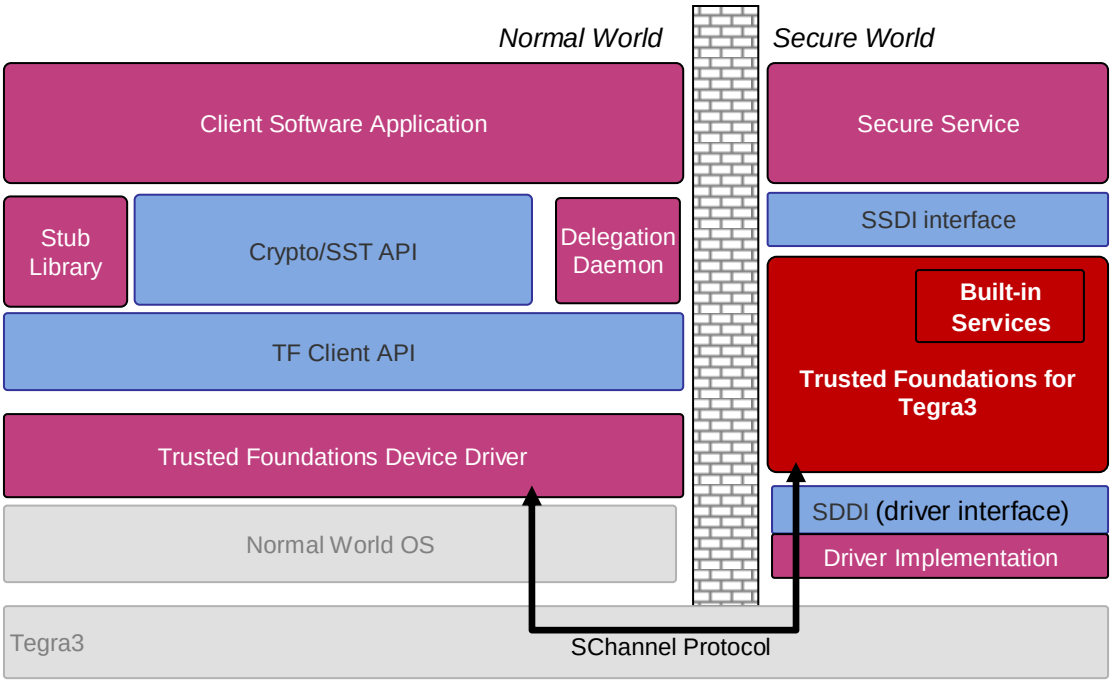

Figure 1 Architecture Overview

The Trusted Foundations for Tegra3 provides a portable Trusted Execution Environment (TEE) for running security-sensitive code. It utilizes TrustZone® to separate the “Secure World” from the “Normal World”:

- The Secure World contains the Trusted Execution Environment that runs Secure Services;
- The Normal World runs Client Applications that access the secure services.

The product includes built-in services that provide off-the-shelf security functionality, such as secure data storage and a cryptographic provider. The product also allows deployment of custom services, which can, for example, implement the heart of a Digital Rights Management scheme.

Refer to Figure 1 for an overview of the overall product architecture.

Intended Audience

This manual is written for those who are integrating Trusted Foundations on a Tegra3-based device. It is assumed that you are an experienced software integrator and that you are familiar with the system level of TrustZone processors.

To know more about how to develop software for the Trusted Foundations, refer to the *Developer Reference Manual* **[DevManual]**.

Using this Manual

This manual is organized as follows:

- Part A, "*Getting Started*" contains general product information. It describes the content of the product package and contains a user guide to the product reference integration on a Tegra3 development platform. This part contains the following chapters:
  - Chapter A1, "*Content of the Product Package*" describes the content of the product package;
  - Chapter A2, "*Using the Reference Integration*" describes an example of integration of the Trusted Foundations on a Tegra3 Platform;
- Part B, "Integration Guide" contains details on the integration of the different components of the Trusted Foundations inside Android. This part contains the following chapters:
  - Chapter B1, "Boot Process" descrives the boot process of a Tergra3 platform with the Trusted Foundations
  - Chapter B2, "Building the secure wolrd components" describes the steps to create/configure a Trusted Foundations with secure services.

  Chapter B3, "Integration into normal world OS" describes the integration of the components in charge of the communication with the Secure world

Part A

GETTING STARTED

This part provides general introductory information about the product. It contains the following chapters:

- Chapter A1, "*Content of the Product Package*" describes the content of the product package;
- Chapter A2, "*Using the Reference Integration*" describes an example of integration of the Trusted Foundations on a Tegra3 Platform;

Chapter A1 CONTENT OF THE PRODUCT PACKAGE

The Trusted Foundations for Tegra3 contains the following components:

- An implementation of the Secure World:
  - for the Tegra3;
  - providing the Secure Service Development Interface to develop secure services running in the Secure World in the C language. All the APIs V3.0 are implemented. Refer to **[DevManual]** for a specification of these APIs;
  - providing the Secure Driver Development Interface to develop secure drivers.
  - containing three built-in secure services that enable the use of the TFSW from the Normal World without any custom programming of the Secure World:
    - External Secure Storage API.
    - Cryptographic API.
    - External Monotonic Counter API
- The Trusted Foundations Development Kit (TF SDK):
  - enabling the development of secure services using the C language and the SSDI interface and supporting RVDS version 4.0,
  - enabling the development of security-demanding applications or libraries for Android,
  - containing a Windows-based simulator that enables the development and debugging of secure services and their associated Normal World software using Microsoft® Visual Studio®.
- A Normal World Operating System Porting Kit that contains the binaries and source code of the components integrated into Android:
  - The Trusted Foundations driver for Android (under GPL),
  - The Trusted Foundations Client API,
  - The Delegation Daemon,
  - The External Cryptographic, Secure Storage and Monotonic Counter library
- A reference integration. See Chapter A2, "*Using the Reference Integration*" for more information about this integration.

The product package contains the following directories:

Directory	Content
`tegra3_secure_world_integration_kit/`	The Secure World Integration, containing the secure world binary and the postlinker
`/tegra3_secure_world_integration_kit/sddk/`	The Secure Driver Development Kit.
`os_porting_kit/`	The OS Porting Kit
`/os_porting_kit/android_porting`	The Android normal world reference integration
`tf_sdk/`	The Trusted Foundations Development Kit.
`/tf_sdk/simulator_ix86_win32`	The Windows simulator. See **[DevManual]** for instructions on how to use the simulator.
`unsupported/tegra3_ref_integration`	Reference integration with pre-compiled and pre-configured binaries for the Tegra3 platform.
`unsupported/python` `unsupported/openssl`	The python and openssl directories contain binary installations of these two tools for Win32 and Linux (ix86).

Chapter A2 USING THE REFERENCE INTEGRATION

This chapter provides the step by step instructions to quickly have the Trusted Foundations running on the Tegra3 platform. This section assumes that the Trusted Foundations has already been included in the Nvidia Tegra3 BSP.

In this chapter, the tag *<PRODUCT_ROOT>* is used to denote the root folder where the product is located and <ANDROID_BUILD_TOP> is used to denote the root folder of the Android source tree.

Update the Trusted Foundations

The Trusted Foundations is provided as a header file to be integrated into the Tegra3 BSP. This file contains a C structure with the binary of the Trusted Foundations inside. From a binary point of view, the Trusted Foundations is included inside the binary of the bootloader.

A version of the Trusted Foundations is provided for reference in the package under <PRODUCT_ROOT>\unsupported\tegra3_ref_integration\tf_include.h.

To use this version, simply copy this header file into the the Tegra3 Bootloader and re-generate the images that will be flashed on the device.

```
> cp <PRODUCT_ROOT>/unsupported/tegra3_ref_integration/tf_include.h
<ANDOID_BUILD_TOP>/vendor/nvidia/proprietary_src/core/system/fastboot/
```

Install Trusted Foundations binaries and libraries into Android OS

To communicate between security demanding applications and the Trusted Foundations, several components are required:

- The TF daemon, in charge of the secure storage. This binary should be present on the Android filesystem under /system/bin. It is considered as an Android service and is started during initialization time by the init.ventana.rc file.
- Other libraries are also required to use secure services of the Trusted Foundations. These libraries should be present on the Android filesystem under /system/lib.
  The libtf_crypto_sst.so is required to use the built-in services of the Trusted Foundations (SST, PKCS11, MTC). This file is provided in the package under <PRODUCT_ROOT>\os_porting_kit\android_porting\tf_crypto_sst\build\release.
  The libsmapi.so,is required for secure services based on the Trusted Foundations API v2.3. This file is provided in the package under:
  <PRODUCT_ROOT>\os_porting_kit\android_porting\tfapi\build\release

How to validate the integration of the Trusted Foundations

The package contains a set of examples, under `<PRODUCT_ROOT>\tf_sdk\examples\`, that can be used to quickly check the Trusted Foundations is up and running. All these examples should be considered usual Android applications and must be installed on the device filesystem under `/system/bin`:

- `tfctrl`, simply requests versioning information about the Trusted Foundations. This simple examples allows to quickly check the Trusted Foundations is up and running
  Here is an example of execution of the `tfctrl` application:

```
> tfctrl list

[Implementation Properties]
 apiDescription  : TFNAB01.01.31965
 commsDescription: TFNAB01.01.0
 TEEDescription  : TFNXB01.01.31965     /Trusted-Logic


[56304B83-5C4E-4428-B99E-605C96AE58D6]
smx.name: SERVICE_SYSTEM
```

- `example_sst`, calls once all the APIs of the secure storage. It allows checking that the TF Daemon is up and running.
  Here is an example of execution of the `example_sst` application:

```
> example_sst

Secure Storage Service Example Application
Copyright (c) 2005-2009 Trusted Logic S.A.

SST: Initialize the service :                        SST_SUCCESS
SST: Create a file :                                 SST_SUCCESS
SST: Write first data block :                        SST_SUCCESS
SST: Write second data block :                       SST_SUCCESS
SST: Checking the EOF :                              SST_SUCCESS
SST: Seeking to the start of the file :              SST_SUCCESS
SST: Read back first data block :                    SST_SUCCESS
SST: Seek to the start of second data block :        SST_SUCCESS
SST: Read back second data block :                   SST_SUCCESS
SST: Remove second data block by truncating :        SST_SUCCESS
SST: Get file pointer position :                     SST_SUCCESS
SST: Get file size :                                 SST_SUCCESS
SST: Remove the file :                               SST_SUCCESS
SST: Terminate the service :                         SST_SUCCESS

Example Summary : SUCCESS
```

- The package contains many more examples that use more complex/specific features. There is one example for each features of the Trusted Foundations product. Please refer to **[DevManual]** for a full documentation on the Trusted Foundations examples.
  In any cases, the two examples above must be considered as the first steps to check an integration of the Trusted Foundations.

Part B

INTEGRATION GUIDE

This document describes the integration of the Trusted Foundations Software on a Tegra3-based device. It contains the following chapters:

- Chapter B1, "*Boot process*" describes the boot steps of a platform running the Trusted Foundations;
- Chapter B2, "*Building secure world components*" describes how to create/configure the Trusted Foundations;
- Chapter B3, "*Integration into normal world OS*" describes the integration of the Trusted Foundation components into the Android OS.

Chapter B1 BOOT PROCESS

For the overall security of the device, it is important that the device implements a Secure Boot process and that the debug interface is controlled. This usually implies that the OEM:

- **burns some key and other ids during the device manufactory,**
- **signs the bootloader and the Trusted Foundations image with a secure boot key,**
- **disables the JTag interface.**

This list is not exhaustive; the OEM should contact Nvidia for further details on how to enable the Secure Boot process and how to configure the hardware at the manufactory to reach the appropriate security level.

The Figure 2 details the boot process of a Tegra3 device with the Trusted Foundations.

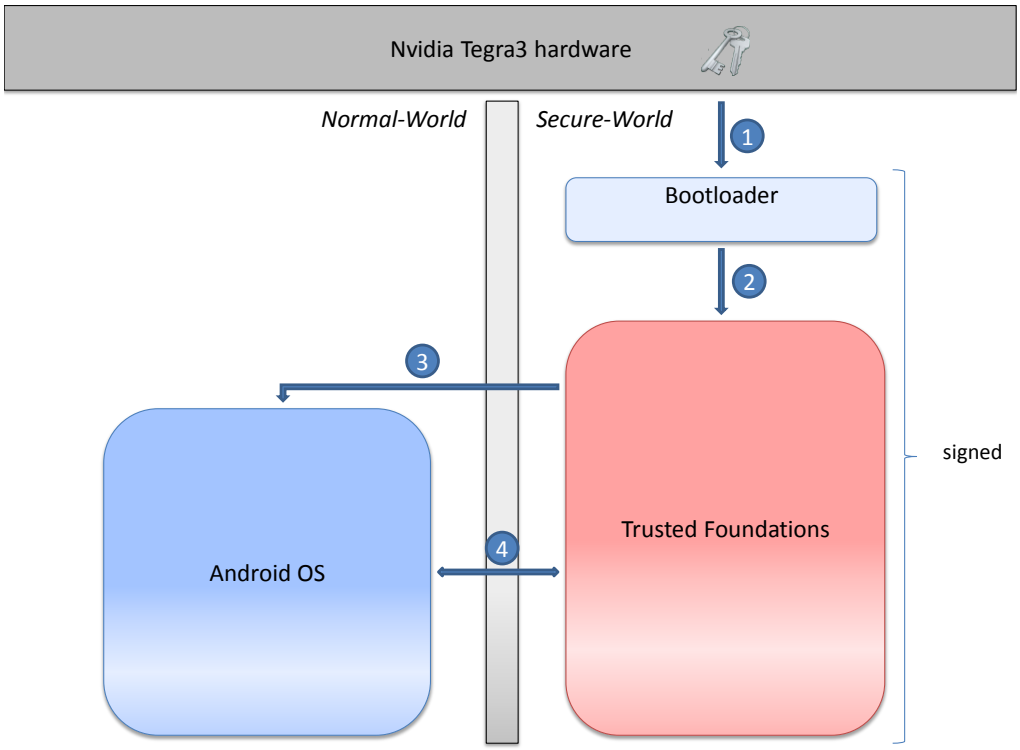

Figure 2 Tegra3 Boot sequence

- Step 1: As part of the secure boot process, the ROM verifies the Bootloader image, copies the Bootloader in memory and executes the Bootloader. By design, on ARM platforms, the Bootlaoder is executed in the Secure-World. The Bootloader prepares the platform.
- Step 2: The Bootloader starts the Trusted Foundations. The Trusted Foundations binary is included into the Bootloader image. Its authenticity is verified by Step 1 along with the Bootloader authenticity. The Trusted Foundations configures the protected memory area and initializes the Secure-World.
- Step 3: The Trusted Foundations jumps to the Normal-World so that the Android OS is started.
- Step 4: Once the Android OS is started, the Trusted Foundations is isolated into the Secure-World through the TrustZone technology. Android can call the Trusted Foundations at any time through the libraries and drivers detailed in the rest of this document.

Chapter B2 BUILDING SECURE WORLD COMPONENTS

The integration of the Trusted Foundations can be divided between

- the **Secure World integration**, which consists of integrating the Trusted Foundations into the secure part of the platform,
- and the **Normal World integration**, which consists of integrating the Trusted Foundations Driver and libraries in the Normal World operating system (Android).

This chapter focus on the steps required to configure and integrate the Trusted Foundations on a Tegra3 based device.

B2.1 CONFIGURING THE TRUSTED FOUNDATIONS

The reference Trusted Foundations binary comes with built-in secure services for cryptography and secure storage. If additional secure services or custom drivers are required, then a specific PC tool called the postlinker must be used to aggregate the Trusted Foundations core binary and the binaries of the secure services into one single Trusted Foundation binary.

Note that the maximum size of the Trusted Foundations after the post-link step is limited. Please refer to section B2.1.5 for details about the maximum size of the Trusted Foundations (after the post-link step).

Besides the role of aggregating custom secure services to the Trusted Foundations Core binary, the postlinker can also be used to inject some specific configuration properties into the Trusted Foundations

The figure below describes the mandatory steps to have a Trusted Foundations with secure services (or custom drivers) inside.

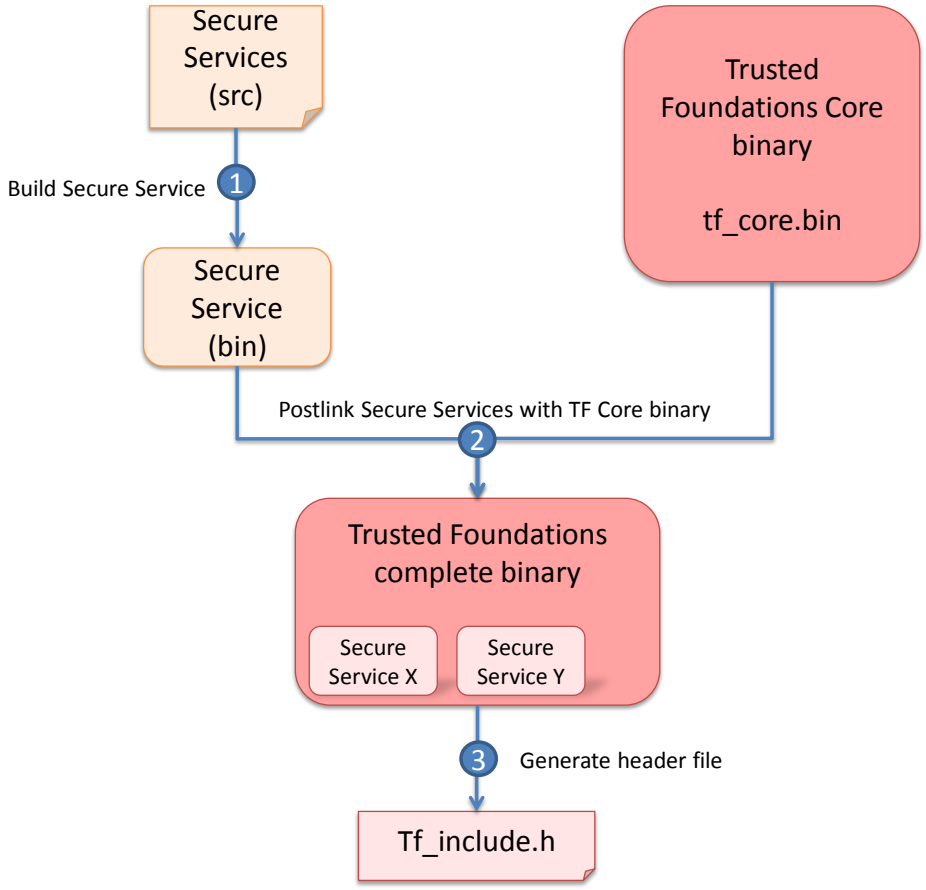

Figure 3 Postlink sequence

- Step 1: Build a secure service or custom driver.
- Step2: Use the postlinker tool to aggregate the secure services and drivers into the Trusted Foundations core binary
- Step3: Transform the Trusted Foundations binary into a C structure contained into a header file that must be compiled with the Bootloader.

For reference a script called `generate_ref_integration.bat` is provided in the Trusted Foundations package, It is an example of how to postlink secure services (or drivers) with the Trusted Foundations. The complete binary created is called `tf_ref_integration.bin`, it includes the Trusted Foundations core binary and all the secure services (or drivers) postlinked.

This script also generates the header file with the Trusted Foundations inside. The file `tf_include.h` is the file that must be included inside the Tegra3 bootloader.

```
# cd %PRODUCT_ROOT%\unsupported\tegra3_ref_integration
# generate_ref_integration.bat
```

For reference a pre-built version of the Trusted Foundations is also provided.

B2.1.1 PREREQUISITE

For integrating the TF the following components must be available:

- TF Core Binary in binary form: this is delivered in the TF package in the Secure World Integration Kit directory
- The mandatory Secure Drivers (sdrv_crypto, sdrv_system). They are delivered in the Trusted Foundations package in the Secure World Integration Kit directory.
- Optionaly, additional Secure Services or Custom Drivers that have been built, thus in binary form and that will be incorporated within the TF.
- Secure-World Configuration File for the Trusted Foundations that is used by the postlinker. The `system_cfg.ini` is provided in the package
- The postlinker tool for linking secure services or secure drivers against the Trusted Foundations and setting some configuration parameters. This tool, `tf_postlinker.exe`, is delivered in the Trusted Foundations package. This is a PC tool.

B2.1.2 SECURE WORLD CONFIGURATION FILE

This section specifies Secure World Configuration File. The postlinker tool uses the system configuration file to personalize the core TF binary for a specific hardware platform.

B2.1.2.1 THE GLOBAL SECTION

Each Secure World Configuration File must have precisely one section named “Global”. The configuration properties in this section affect the whole Trusted Foundations.

B2.1.2.1.1 Identification of the Integration

The following property enables the integrator to configure the Secure World binary with a descriptive string of the integration.

This property is available through the `ssdi.apidescription` implementation property of SSDI and the `sm.devicedescription` implementation property of the TF Client API.

Name	Type	Description
`integration.description`	String (Optional)	A description of the integration. This string must contain only ASCII, non-control characters (0x0020-0x007E). When this string is specified, the version description of the core Trusted Foundations (like “`TFNAB 01.01.`*`<BuildNumber>`*”) is concatenated with the character ‘`/`’ and the specified property. The result is accessible through the TF Client API and SSDI implementation properties. Note that it is guaranteed that the version description of the core does not contain the character ‘`/`’, allowing for easy programmatic delimitation of the two parts

		of the string. This property is optional. If not specified, an empty string is assumed. The implementation limits the number of characters in the `integration.description` property. When it is present, it MUST contain at most 43 ASCII characters.

B2.1.2.1.2 Secure Memory Configuration

The Trusted Foundations needs some RAM space for its execution. The size of the RAM required depends on the features within the Trusted Foundations and the size and complexity of the services to be run.

The Secure World bootloader is responsible for setting up the Trusted Foundations workspace memory. This includes configuring the system controllers, and of static and/or dynamic memory controllers, to correctly map devices into regions of physical memory. It may also include configuration of demarcation devices to designate sub-sections of memory as secure or non-secure. The workspace memory must be configured to be secure (through the use of the TrustZone® hardware), thus protecting against Normal World software attacks, and may also reside on-SoC if some protection against physical attacks is desired.

The Secure World workspace memory must be described in the Secure World Configuration Fileas a list of memory segments. Each segment describes a contiguous range of physical addresses usable by the Trusted Foundations and their memory attributes. In the system configuration file, a segment is identified with a number `<segmentID>` that can be arbitrarily set by the integrator and the following properties must be provided:

Name	Type	Description
`workspace.<segmentID>.address`	Integer (Mandatory)	Physical address of a secure workspace memory segment that can be used by the Trusted Foundations. Must be aligned on a 4KB boundary.
`workspace.<segmentID>.size`	Integer (Mandatory)	Size in bytes of the secure workspace memory segment. Must be a multiple of 4KB.
`workspace.<segmentID>.attributes`	Memory Attributes (Optional)	Memory Attributes of the secure workspace memory segment. Default value is `0b000000` (strongly-ordered)

Option no more available from configuration file. On current implementation, these values are defined at runtime through the Trusted Foundations boot argument.

B2.1.2.1.3 Configuration of the Normal World OS Entry Points and Memory Range

The Trusted Foundations is responsible for booting the Normal World OS. The physical address of the Normal World OS entry point must be configured in the System Configuration File. This is normally the address of the Normal World bootloader but may be the start address of any Normal World code if no bootloader is required.

Name	Type	Description

`normalOS.entryPoint.coldBoot`	Integer (Mandatory)	Physical address in the Normal World memory where the Trusted Foundations must switch to perform a cold boot of the Normal World operating system. This address must be aligned on a 4-byte boundary. The code at this address will be entered in ARM state.

The communication between the two worlds makes use of shared memory that the OS passes by address to the Secure World. It is critical for security that, when the OS sends an address to the Secure World, the Secure World must be able to check that the address does belong to the Normal World and does not point to some Secure World memory or peripherals. There is usually no hardware means to detect this, so this information must be configured by the integrator in the System Configuration File.

The ranges of Normal World memory addresses that can be safely used to share information with the Normal World are described as a list of segments. Each segment describes a contiguous range of physical addresses. This range is allowed to contain holes, i.e. address blocks that do not correspond to actual memory in the memory map. However, no segment may span any physical addresses used by the Secure World.

Each Normal World segment is identifier by a number *<segmentID>* which can be arbitrarily chosen by the integrator. For each segment, the following properties must be configured:

Name	Type	Description
`normalOS.RAM.<segmentID>.address`	Integer (Mandatory)	Start address of a range of physical addresses in which the Normal World memory exists. This memory range may contain holes (regions which are not valid addresses in the physical memory map). Note: the Trusted Foundations uses the information contained in this section to check that a memory reference built by the normal-world driver and passed to the Secure Wolrd does in fact correspond to Normal World memory.
`normalOS.RAM.<segmentID>.size`	Integer (Mandatory)	Size of the Normal World memory segment.

The properties are named "`normalOS.RAM.*`" because, usually, only RAM is shared between the two worlds. However, note that any physical address range can be configured, including peripherals, if necessary.

This version of the product supports up to 8 normal world memory segments.

Option no more available from configuration file. On current implementation, "normalOS.RAM" values are defined at runtime through the Trusted Foundations boot argument (only one memory segment is supported).

B2.1.2.1.4 Authentication Keys

The Trusted Foundations comes with a default cryptographic key used to authenticate client applications.

Integrators should personalize the authentication key by injecting a new key through the postlinker. Otherwise, the default developer key will be used and the integrity of clients cannot be guaranteed as this key is published in the documentation. Note that, even if the integrator uses his own public key, he

must still take special care to ensure the integrity of theft binary on the platform, e.g., using a Secure Boot process.

The client authentication key must be an RSA public key up to 2048-bits in length. It must be represented as an `RSAPublicKey` object as defined in PKCS#1 and encoded using DER.

The DER-encoded key must be put in a file that is provided to the postlinker through the following property:

Name	Type	Description
`clientAuthentication.rootKey`	String	Indicates the path to the file containing the RSA public key for client authentication. This path is relative to the system configuration file. This parameter is optional. If not specified, the default developer key will be used to authenticate clients.

B2.1.2.1.5 Tasks Profiling

This option of the Trusted Foundations enables the display of the current execution task. For each context switch the new task scheduled will be displayed (warning, the tasks profiler option is based on traces mechanism, the overall performance of the system may be affected).

Name	Type	Description
`skernel.task.profiler`	boolean (Optional)	If true, then the scheduler of the Trusted Foundations will print info on the current execution task. Default is false.

B2.1.2.2 SERVICE/DRIVER CONFIGURATION SECTIONS

All the other sections of the Secure World Configuration File are each relative to a specific service or driver. Each service/driver is identified with a UUID. The name of each section must be the string representation of a UUID.

The specification of the properties of a service or driver can be found in the documentation provided by the developer.

Some properties are common to all services/drivers. They are abstractly specified in the following sections.

The service or driver configuration properties come from two different sources:

- The manifest of the service or driver, written by the developer.
- The system configuration file, written by the system integrator.

The postlinker does not make a difference between the two sources of properties. Except when otherwise stated, the properties defined in the service/driver manifest cannot be overridden by the system configuration file. If the same property exists in both the service/driver manifest and in the system configuration file, the postlinker will report an error.

The names of the configuration properties defined in this specification are all prefixed by "`config.s.`" because they are standard configuration properties interpreted by the system. Other properties can be defined in the Secure World Configuration File or service manifest, but they are not interpreted by the system.

B2.1.2.2.1 Memory Segments

Memory segments must be used to configure the physical addresses of peripherals so that they can be made visible in the virtual address spaces of the corresponding drivers.

Each segment is identified by a number, determined by the driver provider, which relates the driver provider and the system integrator:

- The driver uses this identifier to refer to the segment at runtime.
- The integrator uses the identifier to refer to the segment in the Secure World Configuration File and to set the segment address, size, and attributes.

A driver configuration may configure one or more memory segments, according to requirements of the hardware. Some drivers may not require any physical memory segment; in such cases the memory segment properties do not need to be defined.

Each segment is defined by the following properties:

Name	Type	Description
`config.s.mem.<segmentID>.address`	Integer (Optional, depends on the driver)	Physical address of the segment. Usually aligned on a 4KB boundary.
`config.s.mem.<segmentID>.size`	Integer (Mandatory when address is provided)	Number of bytes in the segment. Usually a multiple of 4KB.
`config.s.mem.<segmentID>.attributes`	Memory Attributes (Optional)	Default value is `0b000000` (strongly-ordered)

For each segment:

- The identifier `<segmentID>` is local to the driver and is usually defined by the driver developer in the driver documentation.
- The property `config.s.mem.<segmentID>.address` is usually set by the integrator in the System Configuration File, because only the integrator knows the address of the peripheral. The Trusted Foundations itself makes use of bits [31:12] of the address to set up the Secure World page tables. Bits [11:0] are accessible by the driver at runtime and may be used to denote an offset within a page of registers. They are usually set to 0.
- The property `config.s.mem.<segmentID>.size` is usually provided by the driver developer in the driver's manifest. In such a case, this property does not appear in the System Configuration File. The Trusted Foundations itself uses the segment size to compute the number of pages that must be mapped. The bits [11:0] of the sum `address + size` are therefore not interpreted but may be used by the driver at runtime.
- The property `config.s.mem.<segmentID>.attributes` is optional. If no value is provided, the default value is `0b00000000`, i.e., strongly-ordered (non-cacheable, non-bufferable), which is generally suitable for any peripheral. The integrator may choose to configure the segment as Device (Shared or non-Shared) instead.

Note that a given physical address or a given page can be referred from multiple segments. In such a case, the page will be mapped multiple times in the Secure World virtual memory mapping.

B2.1.2.2.2 Neon Support

Services can activate the support of Neon floating point unit by setting the option `--fpu=VFPv3` at compilation. Additionally a service must define the following property to use Neon:

Name	Type	Description
`config.s.uses_neon`	boolean (Optional)	If true, then the secure service can use Neon. If set to false and if the service tries using Neon, then a Panic occurs. Default is false.

B2.1.2.2.3 Memory Profiling

Each service can activate the memory profiling option. If enabled, the Trusted Foundation prints logs that describe the memory usage done by the service (warning, the memory profiler option is based on traces mechanism, the overall performance of the system may be affected).

Name	Type	Description
`config.s.memory.profiler`	boolean (Optional)	If true, then the secure service will print info on its memory usage Default is false.

To analyze the logs printed by this option, a tool is available in Trusted Foundations package under: `tf_sdk\tools\ tf_memory_profiler.py` can be used to interpret the output printed by a service that has defined the memory profiling option.

B2.1.3 POST LINKING THE TRUSTED FOUNDATIONS

Post linking the Trusted Foundations is required to integrate new Secure Services or Secure Drivers into the Trusted Foundations binary and to configure it.

This is done with a tool called the postlinker (the binary is tf_postliker). This tool runs on PC under Windows 32 (Windows XP) and Linux.

Current version of postlinker supports generation of a Partial binary from the Trusted Foundations core binary and secure services. The post linking step could be repeated several times to aggregate new secure services into the Partial binary until it has become Final one. Only the Final Trusted Foundation binary will be executable.

B2.1.3.1.1 Command line parameters

The postlinker (with “multi-stage” functionality) command line is:

```
tf_postlinker [options] <tf_core.bin> [<service or driver 1> [ <service or
driver 2> ...] ]
```

where the available options are:

```
[-o|--output]  <output file name> Name of the output file to generate
               (mandatory)
[-c|--configuration] <configuration file> Secure-World Configuration File
                     (optional)
[-p|--partial] Generates partial binary (optional)
[-g|--debug]   Generates information useful for debugging (optional)
[--version]    Output the version information
```

- The configuration file is used to set the properties of the Trusted Foundations, as well as Secure Services/Drivers properties. The only case where the configuration file is optional is during the Finalization step, if the properties are unchanged and no Secure Services or Drivers added.

In case of post linking of an existing Partial Trusted Foundations binary, the existing properties (properties of core binary or already postlinked Secure Services) can be updated.

- The output binary generated will depend of the –p or –partial option:
  - If not present: It will generate the Trusted Foundations Final binary (the executable one).
  - If present: the binary generated will be considered as a Partial binary. It means that this binary will have to be re-postlinked (maybe several times) to add Secure Services and/or to finalize it. The Partial Trusted Foundations binary will not be executable, only the Final Trusted Foundations binary will be.

B2.1.3.1.2 Examples of using

Basic post linking functionality is available:

```
# tf_postlinker -o tf_core.bin
                -c system_cfg.ini
                tf_core.bin my_test.srv
```

This directly generates a final Trusted Foundation binary.

Sequence to generate a Partial Trusted Foundation binary (tf_partial.bin) is following:

```
# tf_postlinker -p -o tf_partial.bin
                -c system_cfg.ini
                tf_core.bin my_test.srv
```

Using the Partial binary it is then possible to:

1. Simply finalize it:

```
# tf_postlinker -o tf.bin tf_partial.bin
```

2. Finalize it with customization of the existing properties:

```
# tf_postlinker -o tf.bin -c new_system_cfg.ini tf_partial.bin
```

3. Finalize it and add Secure Services:

```
# tf_postlinker -o tf.bin
                -c new_system_cfg.ini
                tf_partial.bin
                new_test.srv
```

4. Generate a new Partial Trusted Foundations binary:

```
# tf_postlinker -p -o tf_partial_2.bin
                -c new_system_cfg.ini
                tf_partial.bin
                new_test.srv
```

B2.1.4 POSTLINKER PROCESSING

Whenever the postlinker detects an error, it displays an error message on stderr and exits with a non-zero error code. No file is generated unless the whole postlinking process is successful.

1. The postlinker first checks the validity of the command line. If the --version option is provided or if no option at all is provided, then the version information of the tool is displayed on stderr.
2. Then the postlinker checks that first argument is one of the TF core supported by the tool. Note that the postlinker tool is specific to a build of the product and cannot be used with any other builds
3. The postlinker checks the syntax of the Secure-World Configuration File. If a mandatory property is missing or if a property is ill-formed, the processing fails.
4. The postlinker checks that each of the service binary file is compliant with the binary file format specified in this page.
5. The postlinker checks that each UUID listed in the Secure-World Configuration File corresponds to either an in-built service or a file passed on the command-line.
6. The postlinker checks the correctness of the service properties. This depends on the specific file and on the implementation.
7. The postlinker checks that the interface exported by each of the service is supported by the core binary.
8. The postlinker checks that the interfaces imported by each of the service file are supported by the core binary and appropriate for the file interface.
9. The postlinker links together all the files.
10. If the –g or --debug option is specified, then, for each binary passed on the command line, the postlinker creates a file named .robase that contains:
    - The string “--ro-base” followed by a space;
    - The postlinked address in hexadecimal of service‟s RODATA segment. This can be used to debug the service.
11. Finally, the postlinker outputs a summary of the operation containing:
    a. The date and time of the operation;
    b. The MD5 fingerprints of all the inputs files as well as the generated file;
    c. All the global properties;
    d. All the service properties.

B2.1.5 TRUSTED FOUNDATION CODE SIZE LIMIT

On the Tegra3 platform, the size of the complete Trusted Foundations code (after postlinking secure services and custom drivers) must not exceed 1024KBytes.

Chapter B3 INTEGRATION INTO NORMAL WORLD OS

The Trusted Foundations integration into the Normal World OS requires installing:

- A user-land daemon application that takes care in particular of the effective storage of the Trusted Foundations secure storage within the OS file system.
- A driver running in the kernel and that provides a communication stack to the Secure-World.
- User libraries for the different Trusted Foundations APIs exposed in the Normal World, i.e. the Cryptographic, Secure Storage and Monotonic Counter API or custom secure service libraries.

The Figure 4 shows all the Trusted Foundations components:

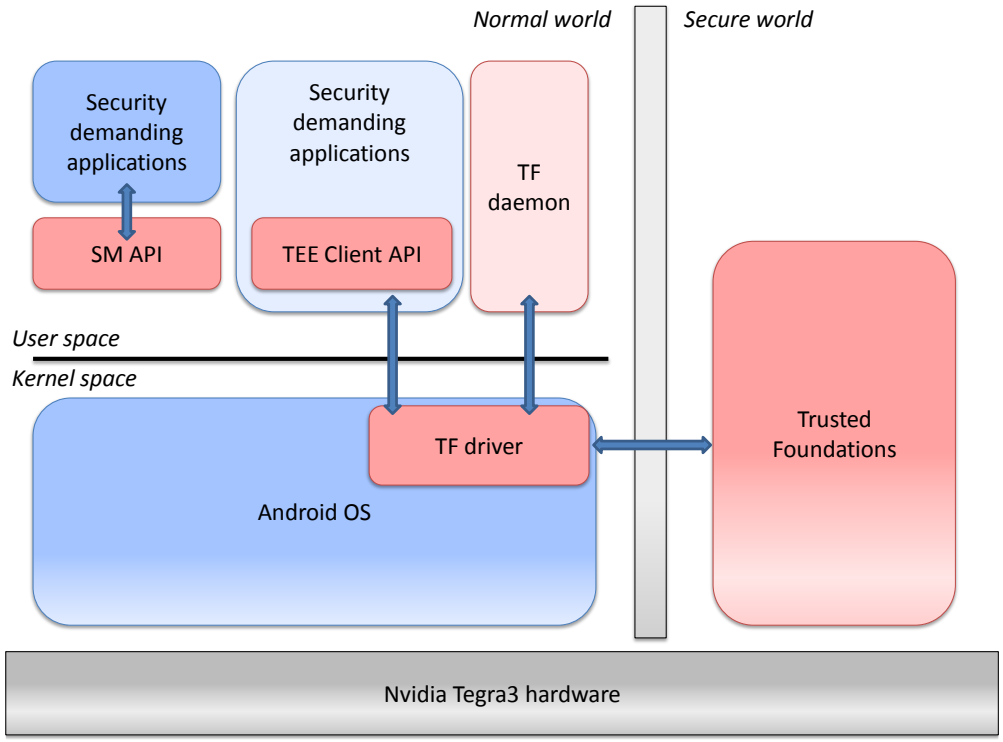

Figure 4 Normal-World architecture.

B3.1 PLATFORM REQUIREMENTS

B3.1.1 BSP CHANGES

Some changes must be applied to the regular BSP in order to integrate the Trusted Foundations. These changes are located into the Tegra3 Bootloader and the Android kernel.

Please contact Nvidia to make sure your BSP is compatible with the Trusted Foundations.

B3.1.2 PLATFORM BOOTLOADER

The Trusted Foundations binary is included in the binary of the Tegra3 Bootloader. The default size of the Trusted Foundations binary on Tegra3 is 1024KBytes.

This means the size of the Bootloader partition in flash memory must be **increased of 1MBytes** to host the Trusted Foundations binary.

Note that the size of the partitions cannot be modified Over the Air once the device has been shipped so this requirement is really important for **installing or upgrading the Trusted Foundations Over The Air**.

B3.1.3 OS MEMORY

By default, the Trusted Foundations is configured **to reserve at startup the last 2MBytes of SDRAM (i.e. external RAM)**. This memory area is dedicated to the Trusted Foundations. It is protected using the TrustZone technology and the Operating System should not attempt to access it. This SDRAM area is dedicated to the Trusted Foundations by the Tegra3 Bootloader.

Figure 5 illustrates the SDRAM memory configuration related to the Secure-World. Due to the alignment constraint, **the start address of these 2MB must be aligned on a 1MB boundary**.

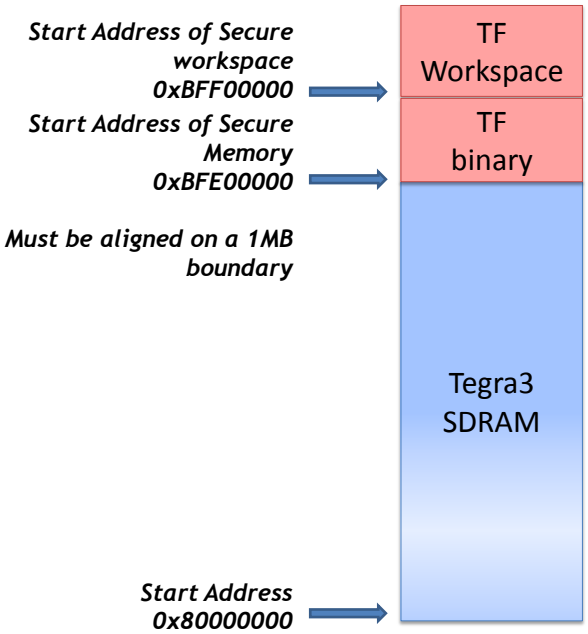

Figure 5: SDRAM Memory Configuration

On Tegra3 platforms reference integration, the Trusted Foundations is configured to use the 2 last MBytes of the memory. It also assumes that the Tegra3 device has 1GB of memory.

However, the Trusted Foundation has no specific limitation regarding the memory configuration: devices can have 2GB of memory or more; the memory reserved for the Trusted Foundations could be set to 15

MB… The sensitive point is to keep the configuration of the OS and of the Trusted Foundation consistent.

For the Trusted Foundations, the memory configuration is defined by file: `system_cfg.ini`. This file is only used by the Trusted Foundations and not by the Android OS or the Tegra3 BSP (ODM data must be set consistently*)*.

Here are the details of the properties that must update:

- `normalOS.RAM.1.size`: default value is 0x3FE00000.
  This indicates the size of the memory available for the Normal World OS (the size of the memory available on the Tegra3 platform without the 2 last MB reserved for the Secure World). For example, on a device with 2GB of RAM available, the value must be 0x7FD00000.
- `workspace.1.address`: 0xBFF00000.
  This indicates the address of the memory workspace used by the Trusted Foundations within the 2MB of reserved memory. This value must be set to `normalOS.RAM.1.size + 0x100000`. In the reference integration, `0x100000` is the maximum size of the TF binary.
  For example, on a device with 2GB of RAM available, the value must be 0xFFE00000.
- `config.s.mem.1.address`: 0xBFFFFF80.
  This indicates a buffer within the 2MB of reserved memory that contains information passed by the Bootloader. This must be set to the last 128 bytes of the 2MB of reserved memory. For a For example, on a device with 2GB of RAM available, the value must be 0xFFEFFF80.

On Tegra3 platform, maximum memory available is not 2GB. The last MB is reserved by the platform. That's why all the values for 2GB configurations are in fact, aligned on 2GB-1MB.

B3.1.4 OS SECURITY

The Trusted Foundations provides an environment to protect assets in particular from any software attacks coming from the Normal World and from the OS. The Trusted Foundations provides also functionality to manage assets that are accessible to the OS and the applications running on top of the OS. For instance, the Trusted Foundations provides functionality such as client authentication and client identification. These features are relying also on the security brought by the OS such as the memory isolation between user and kernel modes and memory isolation between applications. However, specifying how to make and configure the OS as secure as it can be is beyond the scope of this document.

For instance, to illustrate some examples of configuration, in Android, to protect the integrity of shared objects used by client applications, at least the following is recommended:

- not allowing users to have root access;
- ensuring that the dynamically loaded libraries are installed on a read-only file-system that is validated for integrity during the secure boot process.

Furthermore the verification of the full OS integrity is also highly recommended.

B3.2 TF DAEMON

The Trusted Foundations persistent files are built and protected by the Trusted Foundations and stored physically in the device file system through the TF daemon. All the security properties such as confidentiality, integrity, atomicity, anti-rollback and bounding to the device are managed at the Trusted Foundations level. The TF product requires a user-space daemon to be started, which takes care in particular of the effective storage of the TF secure storage within the device file system. This daemon is delivered as a binary application named `tf_daemon`. It must be copied into `<DEVICE_FS>/system/bin/` directory on the device file system.

The TF daemon can be manually, the path to store the persistent data is passed as a command line argument.

```
cd <DEVICE_FS>/system/bin/
chmod 500 tf_daemon
tf_daemon -storageDir /data/tf
```

But, for a safer integration, we recommend to include the start of the tf daemon from one of the initialization scripts of the Android OS by adding the following lines.

```
on boot
  mkdir /data/tf
  service tf_daemon /system/bin/tf_daemon -storageDir /data/tf
  class main
  user root
  group root
  oneshot
```

B3.3 USER LIBRARIES

Software residing in the Normal World using the Trusted Foundations will need access to one or more shared objects / dynamic libraries; these libraries must be installed in a directory accessible by the client applications, `<DEVICE_FS>/system/lib/` for example.

B3.3.1 CORE SECURE SERVICES

The TF product is shipped with some Core Secure Services that provide Secure Storage, Cryptographic and Monotonic Counter Facility via some Normal World APIs. These APIs are implemented into a single shared object called `libtf_crypto_sst.so`.

B3.3.2 TFAPI

Although developers are encouraged to use the static TF Client API, `libtee_client_api_driver.a`, for developing security demanding applications or libraries, the TFAPI shared object is still provided for backward compatibility (`libsmapi.so`).

B3.3.3 CUSTOM SHARED OBJECTS

Finally, any custom shared objects providing stub libraries to 3rd party secure services should also be installed. Please refer to the 3rd party documentation for details.

Part C CUSTOM SECURE DRIVER

This section defines a new type of programmable software component running in the Trusted Foundations: the Custom Driver.

- Chapter C1, "*Developing Secure Drivers*" describes the overall behavior of Custom Secure Drivers;
- Chapter C2, "*Available APIs*" details all the APIs available to develop a Custom Secure Driver.

Chapter C1 DEVELOPING SECURE DRIVERS

A custom driver is very similar to a secure service, except that:

- it is guaranteed to be executed in supervisor mode, whereas secure services may be executed in user mode
- it can therefore directly access hardware. It is provided precisely to host such low-level software whereas a native secure service can only use SSDI and no other resources.
- by default, it is not accessible from normal world clients through the Trusted Foundations API, but only from secure world secure services. For instance, a secure service can send commands to a custom driver via the SXControlxxx functions.

C1.1 PHYSICAL MEMORY MAPPING

Drivers need to access their memory-mapped peripheral's physical addresses. To do so, they can map peripheral's memory into virtual memory, so that when the driver reads or writes in a given mapped address range, it actually accesses the peripheral.

Segments of physical memory are automatically mapped when the first Secure driver instance is created, and are unmapped when the last Secure driver instance is destroyed.

The Secure driver identifies a given memory segment through a logical identifier. Each memory segment is associated with three properties that indicate:

- The physical address of the segment,
- The size of the segment,
- The attributes type of the segment, as defined by the processor.

Note that in this implementation of Trusted Foundations on Nvidia Tegra3 platform, only the default mapping, strongly-ordered, is supported.

At runtime, a Secure driver retrieves the physical address of a segment by fetching the value of the property `config.s.<SegmentID>.address` using service property functions.

C1.2 PROPERTIES

A custom driver can define the same properties as a secure service (see **[DevManual]**).

Properties that are mandatory for a secure service must also be defined for a custom driver. For example, the property `config.s.serviceID` must be defined for a custom driver.

The following additional properties or constraints must be respected by a custom driver:

Property Name	Type	Notes
`config.s.driverType`	String	This property is **mandatory** and **must** be set to "`custom`"
`config.s.multi_instance`	Boolean	These properties must be either: • Not defined • Set to `false`
`config.s.lazy_instantiation`	Boolean	
`config.s.mem.<segmentID>.address`	Integer	The physical address of a mapped memory segment. Must be aligned on a 4KB. This property is **optional**. If this property is not present, the Secure driver cannot map any physical memory. Default mapping is strongly-ordered.
`config.s.mem.<segmentID>.size`	Integer	Number of bytes in the segment. Usually a multiple of 4KB. This property is **mandatory** for each segment address. This means address and size properties must always appear together.
`config.s.custom.allow_nw_clients`	boolean	Whether a Normal-World client can connect to the custom driver.

Note that a custom driver is always mono-instance and instantiated when the system is started. Like a secure service, it may be mono- or multi-session and mono- and multi-command, as defined by the properties `config.s.multi_session` and `config.s.multi_command`.

C1.3 BINARY FILE FORMAT

A custom driver is compiled and linked like a secure service. The format of a custom driver binary file is almost identical to the format of a secure service binary file as defined in the postlinker documentation.

Secure services and custom driver binaries both use the ELF Shared Object file format. The only difference is that the `DT_SONAME` tag must end with the "`.sdrv`" extension in a custom driver instead of the "`.srvx`" extension.

Since the `DT_SONAME` tag usually corresponds to name of the binary file, this means that custom driver binary files usually end with the "`.sdrv`" extension.

Chapter C2 AVAILABLE APIS

A custom driver must implement all entry points of the SSDI-SRVX interface (see **[DevManual]**), except that `SRVXCreate` is replaced by `SDrvCreate` and `SRVXDestroy` is replaced by `SDrvDestroy`.

SDrvCreate

```
S_RESULT SDrvCreate(uint32_t nParam0,
                    uint32_t nParam1);
```

This entry-point is called each time a new driver instance is initialized.

The implementation is free to do any processing required for its initialization. If the driver needs to associate data to the instance, it must allocate this data and call the function `SInstanceSetData`.

The system guarantees that no other entry point of the driver can be called before `SDrvCreate` returns. This may lead to dead lock situations where drivers blocked in the `SDrvCreate` function wait for each other. Therefore, only the most basic initialization operations should be done in `SDrvCreate`. If more complex initialization is required, it should be done in a separate thread.

Parameters

`nParam0-nParam1`: these parameters are not used and are always set to 0 for a custom driver.

Returned Value

`S_SUCCESS` upon successful completion.
`S_ERROR_OUT_OF_MEMORY` if not enough memory is available to complete the operation.
`S_ERROR_UNREACHABLE`: if the underlying hardware is unreachable, either due to a failure or an attack
`S_ERROR_BAD_PARAMETERS`: if the driver properties used to configure the driver are incomplete or ill-formed

Preconditions

The driver has not been instantiated yet.

SDrvDestroy

```
void SDrvDestroy(void);
```

Terminates the instance of the driver.

The implementation must make sure it frees any driver resource including driver's instance data that can be retrieved by calling the function `SInstanceGetData`.

This function does not return any error code. This means that, whatever happens, when the function returns, the framework considers that the instance has been destroyed and it may reclaim the resources consumed by the instance.

Preconditions

No session or operation is pending on the driver instance.

SMemGetVirtual

```
void* SMemGetVirtual(uint32_t nSegmentID);
```

Retrieves the virtual address of a mapped physical memory segment.

The physical memory segment was automatically mapped into virtual memory when the driver was instantiated.

The physical memory segment is identified by the parameter `nSegmentID`. If no segment with this identifier exist, the function returns NULL.

The driver can retrieve the properties of the segment by calling the service properties functions.

This function is **not preemptable**.

Parameters

`nSegmentID`: the logical identifier of the physical memory segment to map.

Returned Value

A pointer to the virtual address of the segment.

`NULL` upon error. In particular, if no segment with this identifier is defined, the functions returns `NULL`.

SMemGetPhysical

```
S_RESULT SMemGetPhysical(void* pVirtual, uint32_t* pnPhysical);
```

Translates a virtual address into its corresponding physical address.

The physical address returned is not a pointer because the driver should not itself dereference a physical address. The expected use is to set up peripherals that can access physical memory.

Note that no access to `pVirtual` is actually performed by the function.

This function is **not preemptable**.

Parameters

`pVirtual`: the virtual address.

`pnPhysical`: a pointer to a 32-bit integer set with the corresponding physical address.

Returned Value

`S_SUCCESS` upon successful completion.

`S_ERROR_ITEM_NOT_FOUND`: the address is not accessible by the calling driver (the memory is not mapped for example)

`S_ERROR_ARM_MEMORY_IS_TCM`: the address points to a TCM. The *`pnPhysical` placeholder is still filled with the physical address but the driver is warned that this address will not be accessible by a bus master, like a DMA controller. The driver must copy the data in some bus accessible region, not use DMA at all in this case, or use the DMA interface of the 1176 to the TCM (unlikely).

SMemFlush

```
S_RESULT SMemFlush(uint32_t  nSegmentID,
                   uint32_t  nOperation);
```

Flushes the given mapped segment out of the data cache.

This function is not supported on current implementations of Trusted Foundations on Nvidia Tegra3 platform.

SMemFlushByAddress

```
S_RESULT SMemFlushByAddress(void* pStartAddress,
                            uint32_t  nLength,
                            uint32_t  nOperation);
```

Flushes the cache for a given address range. This is the same function as `SMemFlush`, but it operates on a range of virtual addresses instead of a memory segment.

The `nOperation` parameter indicates the operation to be performed. It can take one of the following values:

- `S_CACHE_OPERATION_CLEAN`: Writes back any data stored in the cache for the segment and not yet written out to the main memory
- `S_CACHE_OPERATION_INVALIDATE`: Invalidates all the data stored in the cache for this segment. Subsequent read operations will access the main memory. If there is any data stored in the cache for the segment and not yet written out to the main memory, it is discarded
- `S_CACHE_OPERATION_CLEAN_AND_INVALIDATE`: Clean the cache for this segment then invalidate the cache

This function is **not preemptable**.

Conditions under which cache line invalidation is allowed

Normal World client applications and Secure World services may send data to the driver which is not cache line aligned or is not an exact multiple of cache line length. In these conditions the driver cannot safely invalidate the input buffer as it may cause data in the memory that shares part of a cache line with the passed buffer.

If the driver is required to communicate with hardware for these buffers, it must first copy the data to a driver owned buffer which can be safely cleaned and invalidated without causing potential corruption. The driver must ensure that this temporary buffer is correctly locally managed to ensure that this does not cause memory corruption.

Note that it is safe to pass a pointer to Strongly-Ordered or Device memory to the function `SMemFlushByAddress` because no unaligned accesses to the buffer will occur in the function.

Parameters

- `pStartAddress`: the start address of the buffer to flush.
- `nLength`: the length in bytes of the buffer to flush.
- `nOperation`: indicates the operation to be performed.

Panic Reasons

- This function may raise a panic for the following reasons:
- The value of `nOperation` is not valid

Returned Value

`S_SUCCESS` upon successful completion.

Part D IMPLEMENTATION NOTES

This contains the *Implementation Notes* for the Trusted Foundations on Nvidia Tegra3. Throughout the document, this product is simply referred to as “the Implementation”.

This document describes the “implementation-defined” aspects of the APIs defined in the Developer’s Reference Manual **[DevManual]**.

Chapter D1 SECURE WORLD IMPLEMENTATION NOTES

The TFSW binary fully implements the Secure World APIs V3.0, i.e. the APIs specified in **[DevManual]**, chapter B2, "*Secure Service Development Interface*" and in chapter B4, "*Trusted Foundations Cryptographic API*".

D1.1 TRUSTED FOUNDATIONS BOOT ARGUMENTS

On Tegra3 platforms, the platform bootlaoder must start the Trusted Foundations with boot arguments. These arguments, who describe the ”Hardware” configuration of the device, can be viewed as the following C Structure:

```
typedef struct {
   uint32_t   nBootParams;      /* ASCII 'TFBP' (TF Boot Parameters) */
   uint32_t   nDeviceVersion;   /* forwarded to the Normal world OS in r1 */
   uint32_t   nNormalOSArg;     /* forwarded to the Normal world OS r2 */

   uint32_t   nWorkspaceAddress;
   uint32_t   nWorkspaceSize;
   uint32_t   nWorkspaceAttributes ;
   uint32_t   pReserved[16];

   uint8_t    *pParamString;
   uint32_t   nParamStringSize
} TF_BOOT_PARAMS;
```

The format of the argument pParamString is the same than the one used into secure world configuration file:

```
[Global]
normalOS.RAM.1.address: 0x80000000
normalOS.RAM.1.size: 0x3FE00000
normalOS.ColdBoot.PA : 0x80A00800

[770CB4F8-2E9B-4C6F-A538-744E462E150C]
config.s.mem.1.address: 0xBFFFFF80
config.s.mem.1.size: 128

[cb632260-32c6-11e0-bc8e-0800200c9a66]
config.register.aux.value : 0x7E080001
config.register.g.tag_ram_latency : 0x331
config.register.g.data_ram_latency : 0x441
config.register.lp.tag_ram_latency : 0x331
config.register.lp.data_ram_latency : 0x441

[e85b1505-67ac-4d14-8680-4d97c019942d]
config.register.uart.id: 1
```

On current implementations of Trusted Foundations on Tegra3 platform, the arguments supported in the pParamString are exactly those shown above. They are mandatory and have been removed from the secure world configuration file (i.e. they are no more statically customizable on Tegra3 platforms).

D1.2 SSDI IMPLEMENTATION

D1.2.1 SCHEDULING

Processes are scheduled **cooperatively**. A process must therefore call the function `SThreadYield` regularly to give the control to another process if a long-running operation is in progress.

D1.2.2 MEMORY MANAGEMENT

Each process is allocated a virtually contiguous chunk of RAM, whose size is determined by the `config.s.heap_size` property.

D1.2.3 DATE AND TIME API

The value returned by the function `SClockGet` is the value of the current time of the Normal World OS at the time of the last switch from the Normal World to the Secure World. This value is expressed as the number of seconds since January 1, 1970 GMT.

The main source of date for the `SDate` APIs is the same as for `SClockGet`, i.e., the current time provided by the Normal World OS. Each time one of the functions `SDateGet` or `SDateSet` is called, the implementation checks that the main source of date has not rolled back. If a rollback is detected, all service dates are invalidated and later will return the status `S_DATE_STATUS_NEEDS_RESET`.

The value of the implementation property `date.protectionLevel` is the value 1.

D1.2.4 CRYPTOGRAPHIC API

All the mechanisms specified in the Cryptographic API Specification are supported in this product.

D1.2.5 TRACE API

When a service uses the `SLog` API, the format of the trace output is as follows:

```
INFO 00000004 17:34:43.471 P-0021 T-0025 DRM_SAMPLE <MESSAGE>
```

In this example:

- `INFO` is the trace level (it can be also `WARN` or `ERR`)
- `00000004` is an index of the trace line (in decimal)
- `17:34:43.471` is the system-time when the trace was output (Hours:Minutes:Seconds)
- `P-0021` is an identifier of the process that generated the trace
- `T-0025` is an identifier of the thread that generated the trace
- `DRM_SAMPLE` is the name of the calling service. This is either the service UUID or the service property `smx.name` if defined
- *<MESSAGE>* is the service's log message passed into the `SLog` API.

Note for Android

On production version of the Trusted Foundations, the Trace API is fully disabled. It's impossible to output any secure traces.

Note for Windows simulator

The trace outputs are displayed in the shell or command window in which the simulator is started.

D1.2.6 LIMITS ON MEMORY REFERENCES IN STRUCTURED MESSAGES

In this Implementation a secure service may, at most, decode four memory references from a single structured message passed with a command from a client.

Attempting to decode more than four memory references from a single structured message will set the decoder's error state to `S_ERROR_OUT_OF_MEMORY`.

D1.2.7 HARDWARE ACCELERATION FOR FLOATING POINT

The Trusted Foundations for Tegra3 platform supports Neon for floating point hardware acceleration.

Secure services that wish to use Neon have to set their property `config.s.uses_neon` to true. This property is false by default. When the property is set to false and a service tries accessing Neon, a Panic is triggered.

User traps for floating-point exceptions are not supported.

D1.2.8 TRACE DRIVER

On default Trusted Foundations configuration, the Trace driver (mandatory to output traces from a secure service) is not postlinked with the TF Core binary.

When the trace driver is postlinked, traces are directly output to the UART component of the Platform.

D1.2.9 OTF SECURE DRIVER

On default Trusted Foundations configuration, the OTF driver is not postlinked with the TF Core binary.

D1.3 SERVICE MANAGER

As the Implementation does not support downloadable services (no trusted interpreter), the Service Manager only supports the user access mode in SSDI. An attempt to access the Service Manager in manager mode will result in the error `S_ERROR_ACCESS_DENIED`.

D1.4 SECURE FILE-SYSTEM

This Implementation features a single global secure file-system. This file-system provides confidentiality, global integrity, device binding, and optionally protection against replay attacks. The actual storage is delegated to the Normal World file-system.

The secure file-system automatically grows when new data are added. There is no limit regarding the size of the storage or the amount of data that can be stored. The limit is only given by the free space available at the location where the storage files are stored.

The file-system is implemented using fixed-size 'sectors' whose size in bytes can be configured using the global secure configuration property `filesystem.sector.size`. You can configure a sector size of 512, 1024, 2048, or 4096 bytes. If you do not specify a value for the property `filesystem.sector.size`, a default value of 1024 bytes is assumed.

The sector size must stay constant between reboots of the secure world, otherwise, the file-system will be considered corrupted and reformatted.

The implementation of the file-system includes a cache of sectors in secure memory. The size of this cache can be configured using the secure configuration property `filesystem.cache.size`, which is a number of sectors. The cache must always contain at least 3 sectors. A larger secure cache reduces the cryptographic operations incurred by the file-system but consumes more secure memory.

The location and name of the storage files in the Normal World filesystem are specified by arguments given to the TF Daemon command line (see B3.2).

After device boot and when the storage file needs to be accessed at first, the storage file is opened:

- If the storage file does not previously exist, the TF will create an empty storage file.
- If this storage file does exist, the TF opens the storage file. If the corruption is detected while the storage file is being opened, then the TF destroys and recreates an empty storage file (all data stored in this storage file is lost). If the corruption is detected after the storage file has been opened, then the storage file is marked as corrupted: the subsequent function calls accessing this storage file return an error and the storage file will be recreated the next time it is opened (all data stored in this storage file will also be lost).

D1.5 CORE SECURE SERVICE

The Implementation includes a core secure service called the system service that provides cryptography, secure storage and monotonic counter features to the Normal-World.

- System Service (UUID: `56304b83-5c4e-4428-b99e-605c96ae58d6`)

D1.6 MEMORY PROFILING OF SECURE SERVICES

The memory used by a secure service can be monitored by a tool called `tf_memory_profiler`. This tool is available in the Trusted Foundations package under : `<PRODUCT_ROOT>\tf_sdk\tools\.`

It is a command line tool, written in python. It can be started by:

```
> \tf_sdk\tools>python tf_memory_profiler.py
TF Memory Profiler 2.0
Usage: tf_memory_profiler.py [Options] Mode

Mode:
   android            Mode = Android mode is enabled, based on ADB
   win_simu           Mode = Win32 Simulator mode is enabled

Options:
  --version            show program's version number and exit
  -h, --help           show this help message and exit
  --duration=DURATION  Run duration
  --refresh=REFRESH    Refresh period
  --service=SERVICE    Name of service to profile
  --noFreebytes        "Free size" column is turned off
  --noHeapSize         "Heap size" column is turned off
  --noFreepercent      "Free size (%)" column is turned off
  --noLiveblocks       "Live blocks" column is turned off
  --noStackMaxUsage    "Stack max usage" column is turned off
  --noStackTotal       "Stack size" column is turned off
  --noMes              Debug messages are disabled

>
```

For example, to use it with the TF Win32 simulator, type:

```
> \tf_sdk\tools>python tf_memory_profiler.py win_simu
```

And to use it with an Android device, type:

```
> \tf_sdk\tools>python tf_memory_profiler.py android
```

Once started, the information is summarized under 7 columns:

- The name of the service profiled.
- The maximum heap size available for this service
- The memory currently available for this service
- The percentage of the memory available
- The number of memory blocks not yet free
- The maximum stack size available for this service
- The maximum usage of stack for this service

The tf_memory_profiler is based on logs printed by the profiled Secure Service. These logs are controlled by the property: `config.s.memory.profiler` (it must be defined a postlink time).

Android version is based on ADB/Android `dmesg` application to transmit the logs printed by the Secure Service to the tf_memory_profiler (implemented via a polling loop to check if yes or not there are new information to process).

Chapter D2 NORMAL WORLD IMPLEMENTATION NOTES

D2.1 OS PORTING KIT

The Trusted Foundations components for the Android OS comprise:

- The TF Driver for Android under GNU General Public License version 2 **(this is the only component of the Trusted Foundations product package which is delivered under GPL license)**.
- The TF Daemon for Android (eabi format)
- The external secure storage, cryptographic and monotonic counter APIs are provided into a single library. This shared object is compiled for the eabi format.
- The V2-compatible TFAPI library is still provided in eabi format for compatibility.

Note that the new Trusted Foundations Client API is a static library that must be linked with the calling application or library. The Trusted Foundations Client API is provided in the SDK.

Examples of use of the Trusted Foundations Client API and core secure services are provided in the SDK. The examples are provided as source code and build files as well as pre-built binary for the supported Operating System.

D2.2 EXTERNAL CRYPTOGRAPHIC INTERFACE

The crypto core secure service exposing crypto functionality for client applications supports exactly the same mechanisms as the ones exposed by the internal PKCS#11 SSDI Crypto API (see D1.2.4).

D2.3 TRUSTED FOUNDATIONS CLIENT API IN KERNEL

The TF Client API defines a communications API for connecting *Client Applications* running in a rich operating environment with the *Secure Services* running inside the Trusted Foundations execution environment.

On Tegra platforms, the TF Client API is also available for connecting *Kernel drivers* with Secure Services or drivers running inside the Trusted Foundations. The TF Client API available in kernel is similar than the TF Client API available for C*lient Applications.* For more details refer to the *Developer Reference Manual* **[DevManual]**.

D2.4 TRUSTED FOUNDATIONS CLIENT API LOGIN METHODS

The TF Client API defines different login methods that indicate what identity credentials are used to determine access control permissions to functionality provided by, or data stored by, the Secure Service.

This section details how the identity credential is computed for each login types.

TEEC_LOGIN_PUBLIC

No identity is associated with this login type.

TEEC_LOGIN_USER

Note for Android

The identity credential is the application effective user ID (EUID).

Note for Windows simulator

No identity is associated with this login type.

TEEC_LOGIN_GROUP

The calling application must indicate the ID of the group it belongs to as a `uint32_t` in the `connectionData` parameter of the `TEEC_OpenSession` function.

Note for Android

The implementation checks the group provided matches the application effective group ID (eGID) or any of the supplementary group IDs.

Upon successful verification, the identity credential is the group ID.

Note for Windows simulator

The implementation accepts any group ID. The identity credential is the group ID.

TEEC_LOGIN_APPLICATION

Note for Android

The identity credential is the real UID of the application.

Note for Windows simulator

The identity credential is a SHA-1 hash of the concatenation of the fully-qualified name of the calling application and the client IP address. The hash is truncated to 16 bytes.

TEEC_LOGIN_USER_APPLICATION

Note for Android

The identity credential is the real UID and the EUID of the application.

Note for Windows simulator

The identity credential is a SHA-1 hash of the concatenation of the fully-qualified name of the calling application and the client IP address. The hash is truncated to 16 bytes.

TEEC_LOGIN_GROUP_APPLICATION

Note for Android

The calling application must indicate the ID of the group it belongs to as a `uint32_t` in the `connectionData` parameter of the `TEEC_OpenSession` function.

The implementation checks the group provided matches the application effective group ID (eGID) or any of the supplementary group IDs.

Upon successful verification, the identity credential is the real UID and the requested group ID.

Note for Windows simulator

The implementation accepts any group ID. The identity credential is a SHA-1 hash of the concatenation of the fully-qualified name of the calling application, the client IP address and the requested group ID. The hash is truncated to 16 bytes.

TEEC_LOGIN_AUTHENTICATION

The identity credential is the SHA-1 hash of the application binary.

TEEC_LOGIN_PRIVILEGED

Note for Android

The implementation checks that either the application EUID is 0 or the application EGID is 0. No identity is associated with this login type.

Note for Windows simulator

The implementation does not enforce any verification. No identity is associated with this login type.

Part E REFERENCES

[DevManual] Trusted Logic, “Developer Reference Manual (API v3.0)”, CP-2010-RT-533.