User Guide

Release v2.1.0-RC1

ARM CE-OSS

May 10, 2024

OVERVIEW

CHAPTER

ONE

INTRODUCTION

1.1 Trusted Firmware M

Trusted Firmware-M (TF-M) implements the Secure Processing Environment (SPE) for Armv8-M, Armv8.1-M architectures (e.g. the Cortex-M33, Cortex-M23, Cortex-M55, Cortex-M85 processors) and dual-core platforms. It is the platform security architecture reference implementation aligning with PSA Certified guidelines, enabling chips, Real Time Operating Systems and devices to become PSA Certified.

TF-M relies on an isolation boundary between the Non-secure Processing Environment (NSPE) and the Secure Processing Environment (SPE). It can but is not limited to using the Arm TrustZone technology on Armv8-M and Armv8.1-M architectures. In pre-Armv8-M architectures physical core isolation is required.

TF-M consists of:

- Secure Boot to authenticate NSPE and SPE images
- TF-M Core for controlling the isolation, communication and execution within SPE and with NSPE
- Crypto, Internal Trusted Storage (ITS), Protected Storage (PS), Firmware Update and Attestation secure services

TF-M implements PSA-FF-M defined IPC and SFN mechanisms to allow communication between isolated firmware partitions. TF-M is highly configurable allowing users to only include the required secure services and features. Project provides *Base Configuration* build with just TF-M core and platform drivers and 4 predefined configurations known as *TF-M Profiles*. TF-M Profiles or TF-M base can be configured to include required services and features as described in the *Configuration* section.

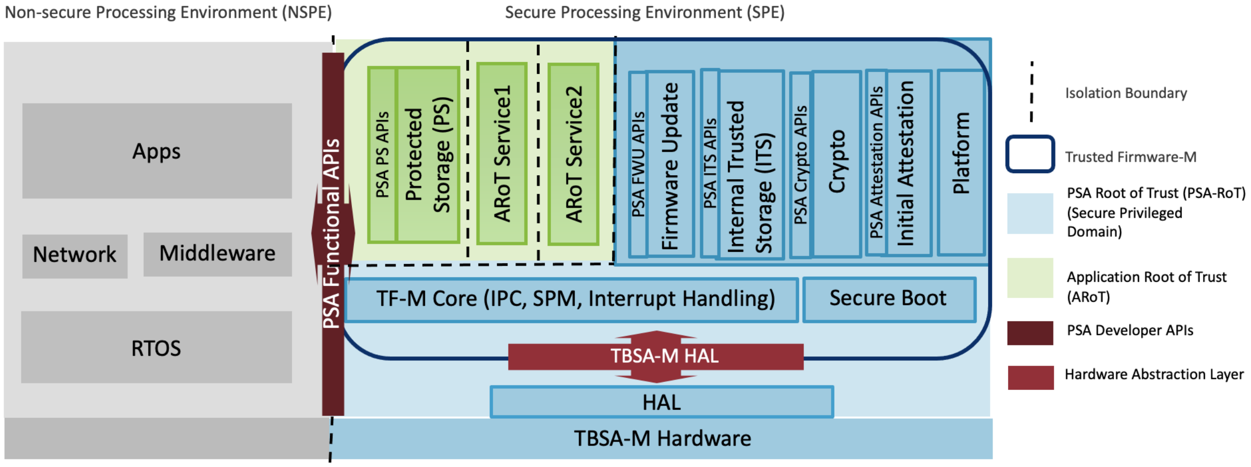


Figure 1:: FF-M compliant design with TF-M

Applications and Libraries in the Non-secure Processing Environment can utilize these secure services with a standardized set of PSA Functional APIs. Applications running on Cortex-M devices can leverage TF-M services to ensure secure connection with edge gateways and IoT cloud services. It also protects the critical security assets such as sensitive data, keys and certificates on the platform. TF-M is supported on several Cortex-M based *Microcontrollers* and Real Time Operating Systems (RTOS).

Terms `TFM` and `TF-M` are commonly used in documents and code and both refer to `Trusted Firmware M.` *Glossary* has the list of terms and abbreviations.

1.1.1 Repositories

TF-M is comprised of multiple repositories that supplement each other in making the project both customisable and maintainable.

Table 1:: TF-M Repositories

Repository	Description
trusted-firmware-m	Software implementation of TF-M with documentation and essential tools
tf-m-tests	Tests that focus on the functionalities of TF-M components
tf-m-tools	Non essential tools used for testing and verification of TF-M
tf-m-extras	Extension of the main repository to host examples, demonstrations, third-party modules etc

1.2 License

The software is provided under a BSD-3-Clause *License*. Contributions to this project are accepted under the same license with developer sign-off as described in the *Contributing Guidelines*.

This project contains code from other projects as listed below. The code from external projects is limited to `bl2`, `lib` and `platform` folders. The original license text is included in those source files.

- The `bl2` folder contains files imported from MCUBoot project and the files have Apache 2.0 license.
- The `lib/ext` folder may contain 3rd party projects and files with diverse licenses. Here are some that are different from the BSD-3-Clause and may be a part of the runtime image. The source code for these projects is fetched from upstream at build time only.
  - `CMSIS_5` - Apache 2.0 license
  - `mbedcrypto` - Apache 2.0 license MbedTLS
  - `mcuboot` - Apache 2.0 license MCUBoot
  - `qcbor` - Modified BSD-3-Clause license
  - `tf-m-extras` - Set of additional components. Please check individually in tf-m-extras repository
- The `platform` folder currently contains platforms support imported from the external project and the files may have different licenses.

1.3 Supported Platforms

- Cortex-M85 system:
  - Corstone-310 Ethos-U55/U65 Ecosystem FVP.
  - FPGA image loaded on MPS3 board (AN555).
- Cortex-M55 system:
  - FPGA image loaded on MPS3 board (AN547).
  - FPGA image loaded on MPS3 board (AN552).
  - Corstone-300 Ethos-U55/U65 Ecosystem FVP.
- Cortex-M33 system:
  - FPGA image loaded on MPS2 board (AN521).
  - Fast model FVP_MPS2_AEMv8M (AN521).
  - Musca-B1 test chip board.
  - Musca-S1 test chip board.
  - FPGA image loaded on MPS3 board (AN524).
  - LPC55S69.
  - STM32H573I-DK.
  - B-U585I-IOT02A.
  - NUCLEO L552ZE Q.
  - DISCO L562QE.
  - nRF9160 DK (nordic_nrf/nrf9160dk_nrf9160).
  - nRF5340 DK (nordic_nrf/nrf5340dk_nrf5340_cpuapp).
  - BL5340 DVK (lairdconnectivity/bl5340_dvk_cpuapp).
- Cortex-M23 system:
  - FPGA image loaded on MPS2 board (AN519).
  - M2351.
  - M2354.
- Dual Core Cortex-M system:
  - PSoc64.
- Runtime Security Engine (RSE):
  - *RSE*.

The document *Platform Deprecation and Removal* lists the deprecated platforms, removed from the upstream.

The document *Supported Platforms* lists the details.

1.4 Release Notes and Process

The *Release Cadence and Process* provides release cadence and process information.

The *Releases* provides details of major features of the release and platforms supported.

1.5 Feedback and Support

For this release, feedback is requested via email to tf-m@lists.trustedfirmware.org.

A bi-weekly technical forum is available for discussion on any technical topics online. Welcome to join TF-M Forum.

CHAPTER

TWO

GETTING STARTED GUIDES

2.1 First Things First

2.1.1 Prerequisite

Trusted Firmware M provides a reference implementation of platform security architecture reference implementation aligning with PSA Certified guidelines. It is assumed that the reader is familiar with specifications can be found at Platform Security Architecture Resources.

The current TF-M implementation specifically targets TrustZone for ARMv8-M so a good understanding of the v8-M architecture is also necessary. A good place to get started with ARMv8-M is developer.arm.com.

2.1.2 Build and run instructions

Trusted Firmware M source code is available on git.trustedfirmware.org.

To build & run TF-M:

- Follow the this guide to set up and check your environment.
- Follow the *Build instructions* to compile and build the TF-M source.
- Follow the *Run TF-M examples on Arm platforms* for information on running the example.

To port TF-M to a another system or OS, follow the *OS Integration Guide*

Contributing Guidelines contains guidance on how to contribute to this project.

2.2 Set up build environments

TF-M officially supports a limited set of build environments and setups. In this context, official support means that the environments listed below are actively used by team members and active developers, hence users should be able to recreate the same configurations by following the instructions described below. In case of problems, the TF-M team provides support only for these environments, but building in other environments can still be possible.

The following environments are supported:

Linux

Windows

1. version supported:

   Ubuntu 18.04 x64+

2. install dependencies:

```
sudo apt-get install -y git curl wget build-essential libssl-dev python3 \
python3-pip cmake make
```

3. verify cmake version:

```
cmake --version
```

Note: Please download cmake 3.15 or later version from https://cmake.org/download/.

4. add CMake path into environment:

```
export PATH=<CMake path>/bin:$PATH
```

1. version supported:

   Windows 10 x64

2. install dependecies:

- Git client latest version (https://git-scm.com/download/win)
- CMake (native Windows version)
- GNU make (http://gnuwin32.sourceforge.net/packages/make.htm)
- Python3 (native Windows version) and the pip package manager (from Python 3.4 it's included)

3. add CMake path into environment:

```
set PATH=<CMake_Path>\bin;%PATH%
```

2.3 Install python dependencies

Clone the TF-M source code, and then install the TF-M's additional Python dependencies.

Linux

Windows

1. get the TF-M source code:

```
git clone https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git
```

2. TF-M's `tools/requirements.txt` file declares additional Python dependencies. Install them with `pip3`:

```
pip3 install --upgrade pip
cd trusted-firmware-m
pip3 install -r tools/requirements.txt
```

1. get the TF-M source code:

```
git clone https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git
```

2. TF-M's `tools/requirements.txt` file declares additional Python dependencies. Install them with `pip3`:

```
cd trusted-firmware-m
pip3 install -r tools\requirements.txt
```

2.4 Install a toolchain

To compile TF-M code, at least one of the supported compiler toolchains have to be available in the build environment. The currently supported compiler versions are:

- Arm Compiler v6.13 ~ v6.14, v6.18+

  Linux

  Windows

  - Download the standalone packages from here.
  - Add Arm Compiler into environment:

    ```
    export PATH=<ARM_CLANG_PATH>/bin:$PATH
    export ARM_PRODUCT_PATH=<ARM_CLANG_PATH>/sw/mappings
    ```

  - Configure proper tool variant and license.
  - Download the standalone packages from here.
  - Add Arm Compiler into environment:

    ```
    set PATH=<ARM_CLANG_PATH>\bin;%PATH%
    set ARM_PRODUCT_PATH=<ARM_CLANG_PATH>\sw\mappings
    ```

  - Configure proper tool variant and license.

  Note: Arm compiler v6.15 ~ v6.17 may cause MemManage fault. This defect has been fixed since Arm compiler v6.18. See [SDCOMP-59788] in Armclang v6.18 release note for details.

- GNU Arm compiler v7.3.1+

  Linux

  Windows

  - Download the GNU Arm compiler from here.
  - Add GNU Arm into environment:

    ```
    export PATH=<GNU_ARM_PATH>/bin:$PATH
    ```

  - Download the GNU Arm compiler from here.
  - Add GNU Arm into environment:

    ```
    set PATH=<GNU_ARM_PATH>\bin;%PATH%
    ```

  Note: GNU Arm compiler version *10-2020-q4-major* has an issue in CMSE support. The bug is reported in here. Select other GNU Arm compiler versions instead.

- IAR Arm compiler v8.42.x, v8.50.x

  Linux

  Windows

  - Download IAR build tools from here.
  - Add IAR Arm compiler into environment:

    ```
    export PATH=<IAR_COMPILER_PATH>/bin:$PATH
    ```

  - Download IAR build tools from here.
  - Add IAR Arm compiler into environment:

    ```
    set PATH=<IAR_COMPILER_PATH>\bin;%PATH%
    ```

2.5 Build AN521 regression sample

Here, we take building TF-M for AN521 platform with regression tests using GCC as an example:

Linux

Windows

Get the TF-M tests source code:

```
git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
```

Build SPE and NSPE.

```
cd </tf-m-tests/tests_reg>
cmake -S spe -B build_spe -DTFM_PLATFORM=arm/mps2/an521 -DCONFIG_TFM_SOURCE_PATH=<TF-M␣
↪source dir absolute path> \
      -DCMAKE_BUILD_TYPE=Debug -DTFM_TOOLCHAIN_FILE=<TF-M source dir absolute path>/
↪toolchain_GNUARM.cmake \
      -DTEST_S=ON -DTEST_NS=ON \
cmake --build build_spe -- install

cmake -S . -B build_test -DCONFIG_SPE_PATH=<tf-m-tests absolute path>/tests_reg/build_
↪spe/api_ns \
      -DCMAKE_BUILD_TYPE=Debug -DTFM_TOOLCHAIN_FILE=<tf-m-tests absolute path>/tests_reg/
↪build_spe/api_ns/cmake/toolchain_ns_GNUARM.cmake
cmake --build build_test
```

Important: Use “/” instead of “\” when assigning Windows paths to CMAKE variables, for example, use “c:/build” instead of “c:\\build”.

Get the TF-M tests source code:

```
git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
```

Build SPE and NSPE.

```
cd </tf-m-tests/tests_reg>
cmake -G"Unix Makefiles" -S spe -B build_spe -DTFM_PLATFORM=arm/mps2/an521 -DCONFIG_TFM_
↪SOURCE_PATH=<TF-M source dir absolute path> \
      -DCMAKE_BUILD_TYPE=Debug -DTFM_TOOLCHAIN_FILE=<TF-M source dir absolute path>/
↪toolchain_GNUARM.cmake \
      -DTEST_S=ON -DTEST_NS=ON \
cmake --build build_spe -- install

cmake -G"Unix Makefiles" -S . -B build_test -DCONFIG_SPE_PATH=<tf-m-tests absolute path>/
↪tests_reg/build_spe/api_ns \
      -DCMAKE_BUILD_TYPE=Debug -DTFM_TOOLCHAIN_FILE=<tf-m-tests absolute path>/tests_reg/
↪build_spe/api_ns/cmake/toolchain_ns_GNUARM.cmake
cmake --build build_test
```

Note: The latest Windows support long paths, but if you are less lucky then you can reduce paths by moving the build directory closer to the root by changing the `-B` option of the commands, for example, to `C:\build_spe` and `C:\build_test` folders.

2.6 Run AN521 regression sample

Run the sample code on SSE-200 Fast-Model, using FVP_MPS2_AEMv8M provided by Arm Development Studio.

Note: Arm Development Studio is not essential to develop TF-M, you can skip this section if don't want to try on Arm develop boards.

Linux

Windows

1. install Arm Development Studio to get the fast-model.

   Download Arm Development Studio from here.

2. Add `bl2.axf` and `tfm_s_ns_signed.bin` to symbol files in Debug Configuration menu.

```
<DS_PATH>/sw/models/bin/FVP_MPS2_AEMv8M  \
--parameter fvp_mps2.platform_type=2 \
--parameter cpu0.baseline=0 \
--parameter cpu0.INITVTOR_S=0x10000000 \
--parameter cpu0.semihosting-enable=0 \
--parameter fvp_mps2.DISABLE_GATING=0 \
--parameter fvp_mps2.telnetterminal0.start_telnet=1 \
--parameter fvp_mps2.telnetterminal1.start_telnet=0 \
--parameter fvp_mps2.telnetterminal2.start_telnet=0 \
--parameter fvp_mps2.telnetterminal0.quiet=0 \
--parameter fvp_mps2.telnetterminal1.quiet=1 \
--parameter fvp_mps2.telnetterminal2.quiet=1 \
--application cpu0=<build_spe>/api_ns/bin/bl2.axf \
--data cpu0=<build_test>/tfm_s_ns_signed.bin@0x10080000
```

Note: The log is output to telnet by default. It can be also redirected to stdout by adding the following parameter.

```
--parameter fvp_mps2.UART0.out_file=/dev/stdout
```

To automatically terminate the fast-model when it finishes running, you can add the following parameters:

```
--parameter fvp_mps2.UART0.shutdown_on_eot=1
```

1. install Arm Development Studio to get the fast-model.

   Download Arm Development Studio from here.

2. Add `bl2.axf` and `tfm_s_ns_signed.bin` to symbol files in Debug Configuration menu.

```
<DS_PATH>\sw\models\bin\FVP_MPS2_AEMv8M  \
--parameter fvp_mps2.platform_type=2 \
--parameter cpu0.baseline=0 \
--parameter cpu0.INITVTOR_S=0x10000000 \
--parameter cpu0.semihosting-enable=0 \
--parameter fvp_mps2.DISABLE_GATING=0 \
--parameter fvp_mps2.telnetterminal0.start_telnet=1 \
--parameter fvp_mps2.telnetterminal1.start_telnet=0 \
--parameter fvp_mps2.telnetterminal2.start_telnet=0 \
--parameter fvp_mps2.telnetterminal0.quiet=0 \
--parameter fvp_mps2.telnetterminal1.quiet=1 \
--parameter fvp_mps2.telnetterminal2.quiet=1 \
--application cpu0=<build_spe>/api_ns/bin/bl2.axf \
--data cpu0=<build_test>/tfm_s_ns_signed.bin@0x10080000
```

Note: To automatically terminate the fast-model when it finishes running, you can add the following parameters:

```
--parameter fvp_mps2.UART0.shutdown_on_eot=1
```

After completing the procedure you should see the following messages on the DAPLink UART (baud 115200 8n1):

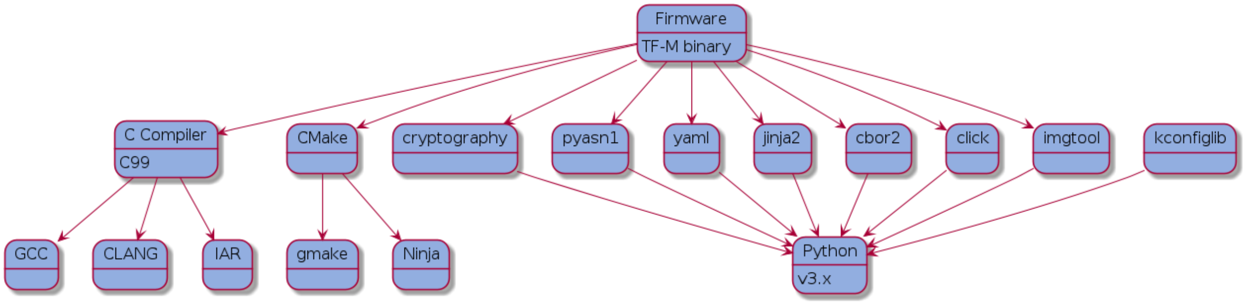
```
...
#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_S_PS_TEST_1XXX)...
> Executing 'TFM_S_PS_TEST_1001'
Description: 'Set interface'
TEST: TFM_S_PS_TEST_1001 - PASSED!
> Executing 'TFM_S_PS_TEST_1002'
Description: 'Set interface with create flags'
TEST: TFM_S_PS_TEST_1002 - PASSED!
> Executing 'TFM_S_PS_TEST_1003'
Description: 'Set interface with NULL data pointer'
TEST: TFM_S_PS_TEST_1003 - PASSED!
> Executing 'TFM_S_PS_TEST_1005'
Description: 'Set interface with write once UID'
TEST: TFM_S_PS_TEST_1005 - PASSED!
....
```

2.7 Tool & Dependency overview

To build the TF-M firmware the following tools are needed:

- C compiler of supported toolchains
- CMake version 3.15 or later
- Git
- gmake, aka GNU Make
- Python v3.x
- a set of python modules listed in `tools/requirements.txt`

2.7.1 Dependency chain



Next steps

Here are some next steps for exploring TF-M:

- Detailed *Build instructions*.
- *IAR Build instructions*.
- Try other *Samples and Demos*.
- *Documentation generation*.

CHAPTER

THREE

SECURITY

3.1 Trusted Firmware-M Generic Threat Model

3.1.1 Introduction

This document introduces a generic threat model of Trusted Firmware-M (TF-M). This generic threat model provides an overall analysis of TF-M implementation and identifies general threats and mitigation.

There is also a dedicated document for physical attacks mitigations which can be found *here*.

Note: If you think a security vulnerability is found, please follow Trustedfirmware.org [Security-Incident-Process] to contact TF-M security team.

Scope

TF-M supports diverse models and topologies. It also implements multiple isolation levels. Each case may focus on different target of evaluation (TOE) and identify different assets and threats. TF-M implementation consists of several secure services, defined as Root of Trust (RoT) service. Those RoT services belong to diverse RoT (Application RoT or PSA RoT) and access different assets and hardware. Therefore each RoT service may require a dedicated threat model.

The analysis on specific models, topologies or RoT services may be covered in dedicated threat model documents. Those threat models are out of the scope of this document.

Methodology

The threat modeling in this document follows the process listed below to build up the threat model.

- Target of Evaluation (TOE)
- Assets identification
- Data Flow Diagram (DFD)
- Threats Prioritization
- Threats identification

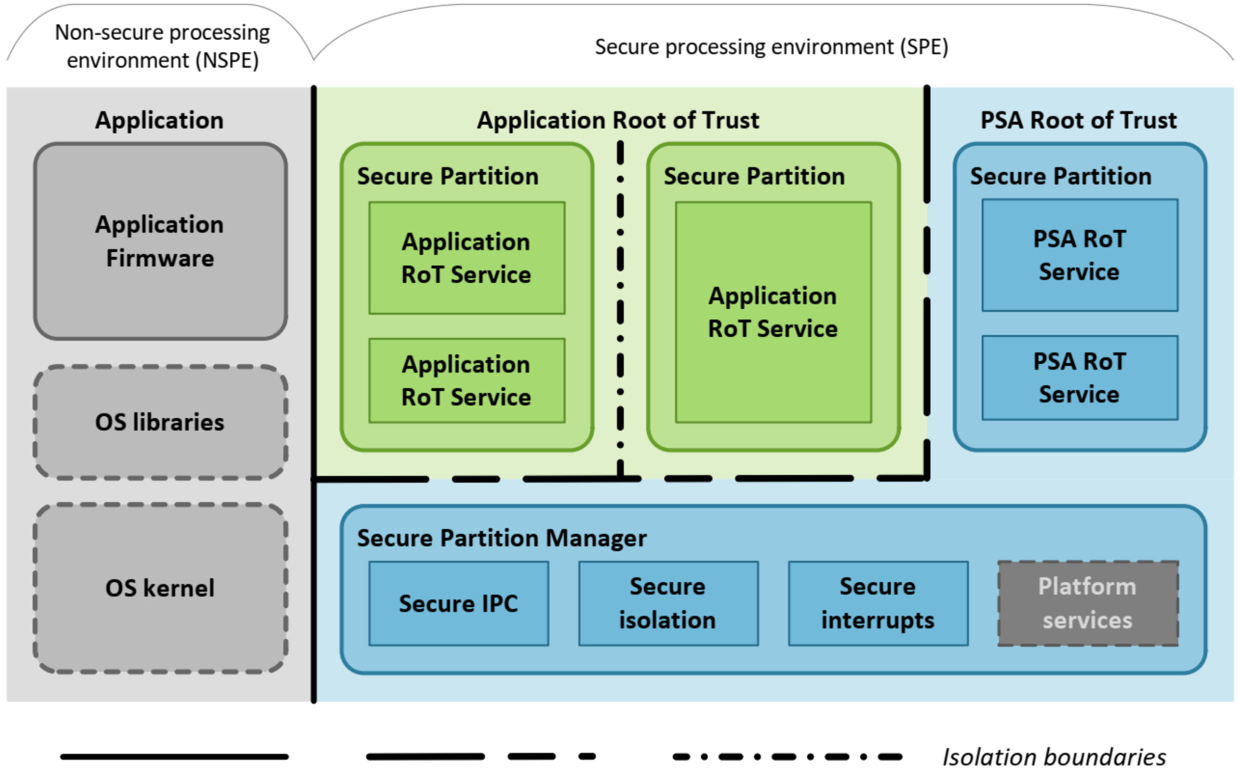
TOE is the entity on which threat modeling is performed. The logic behind this process is to firstly investigate the TOE which could be a system, solution or use case. This first step helps to identify the assets to be protected in TOE.

According to TOE and assets, Trust Boundaries can be determined. The Data Flow Diagram (DFD) across Trust Boundaries is then defined to help identify the threats.

Those threats should be prioritized based on a specific group of principals and metrics. The principals and metrics should also be specified.

3.1.2 Target of Evaluation

A typical TF-M system diagram from a high-level overview is shown below. TF-M is running in the Secure Processing Environment (SPE) and NS software is running in Non-secure Processing Environment (NSPE). For more details, please refer to Platform Security Architecture Firmware Framework for M (FF-M) [FF-M] and FF-M 1.1 Extensions [FF-M 1.1 Extensions]_.



The TOE in this general model is the SPE, including TF-M and other components running in SPE.

The TOE can vary in different TF-M models, RoT services and usage scenarios. Refer to dedicated threat models for the specific TOE definitions.

3.1.3 Asset identification

In this threat model, assets include the general items listed below:

- Hardware Root of Trust data, e.g.
  - Hardware Unique Key (HUK)
  - Root authentication key
  - Other embedded root keys
- Software RoT data, e.g.
  - Secure Partition Manager (SPM) code and data

  - Secure partition code and data
  - NSPE data stored in SPE
  - Data generated in SPE as requested by NSPE
- Availability of entire RoT service
- Secure logs, including event logs

Assets may vary in different use cases and implementations. Additional assets can be defined in an actual usage scenario and a dedicated threat model.

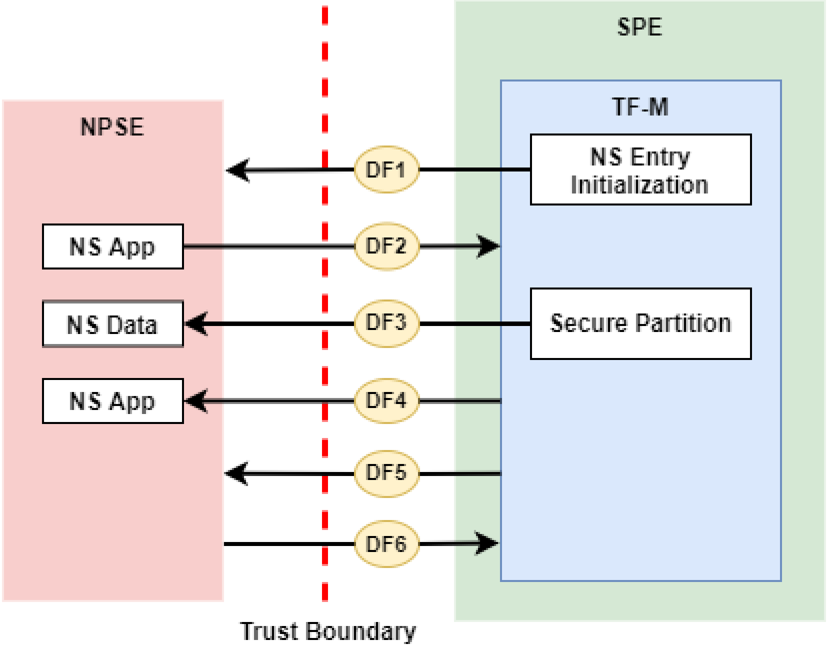
For example, in a network camera use case, the following data can be defined as assets too:

- Certificate for connecting to cloud
- Session keys for encryption/decryption in the communication with cloud
- Keys to encrypt/decrypt the videos and photos

3.1.4 Data Flow Diagram

The Trust Boundary isolates SPE from NSPE, according to the TOE definition in *Target of Evaluation*. The Trust Boundary mapped to block diagram is shown in the figure below. Other modules inside SPE stay in the same TOE as TF-M does.

Valid Data flows across the Trust Boundary are also shown in the figure below. This threat model only focuses on the data flows related to TF-M.



More details of data flows are listed below.

Table 2:: TF-M Data Flows between NSPE and SPE

Data flow	Description
`DF1`	TF-M initializes NS entry and activates NSPE. • On single Armv8-M core platforms, TF-M will hand over the control to Non-secure state. • On dual-cpu platforms, Secure core starts NS core booting.
`DF2`	NSPE requests TF-M RoT services. NSPE requests RoT services via PSA Client APIs defined in [FF-M]. In single Armv8-M core scenarios, SG instruction is executed in Non-secure Callable region to trigger a transition from Non-secure state to Secure state. On dual-cpu platforms, non-secure core sends PSA Client calls to secure core via mailbox.
`DF3`	Secure Partitions fetch input data from NS and write back output data to NS. As required in [FF-M], Secure Partitions should not directly access NSPE memory. Instead, RoT services relies on TF-M SPM to access NSPE memory.
`DF4`	TF-M returns RoT service results to NSPE after NS request to RoT service is completed. In single Armv8-M core scenarios, it also trigger a transition from Secure state back to Non-secure state. On dual-cpu platforms, secure core returns the result to non-secure core via mailbox.
`DF5`	Non-secure interrupts preempt SPE execution in single Armv8-M core scenarios.
`DF6`	Secure interrupts preempt NSPE execution in single Armv8-M core scenarios.

Note: All the other data flows across the Trusted Boundary besides the valid ones mentioned above should be prohibited by default. Proper isolation must be configured to prevent NSPE directly accessing SPE.

Threats irrelevant to data flows in *TF-M Data Flows between NSPE and SPE* may be specified in *Miscellaneous threats*.

Data flows inside SPE (informative)

Since all the SPE components stay in the TOE within the same Trust Boundary in this threat model, the data flows between SPE components are not covered in this threat model. Instead, those data flows and corresponding threats will be identified in the dedicated threat model documents of TF-M RoT services and usage scenarios.

Those data flows inside SPE include following examples:

- Data flows between TF-M and BL2
- Data flows between RoT services and SPM
- Data flows between RoT services and corresponding secure hardware and assets, such as secure storage device, crypto hardware accelerator and Hardware Unique Key (HUK).

3.1.5 Threat identification

Threat priority

Threat priority is indicated by the score calculated via Common Vulnerability Scoring System (CVSS) Version 3.1 [CVSS]. The higher the threat scores, the greater severity the threat is with and the higher the priority is.

CVSS scores can be mapped to qualitative severity ratings defined in CVSS 3.1 specification [CVSS_SPEC]. This threat model follows the same mapping between CVSS scores and threat priority rating.

As a generic threat model, this document focuses on *Base Score* which reflects the constant and general severity of a threat according to its intrinsic characteristics.

The *Impacted Component* defined in [CVSS_SPEC] refers to the assets listed in *Asset identification*.

Threats and mitigation list

This section lists generic threats and corresponding mitigation, based on the the analysis of data flows in *Data Flow Diagram*.

Threats are identified following `STRIDE` model. Please refer to [STRIDE] for more details.

The field `CVSS Score` reflects the threat priority defined in *Threat priority*. The field `CVSS Vector String` contains the textual representation of the CVSS metric values used to score the threat. Refer to [CVSS_SPEC] for more details of CVSS vector string.

Note: A generic threat may have different behaviors and therefore require different mitigation, in diverse TF-M models and usage scenarios.

This threat model document focuses on general analysis of the following threats. For the details in a specific configuration and usage scenario, please refer to the dedicated threat model document.

NS entry initialization

This section identifies threats on `DF1` defined in *Data Flow Diagram*.

Table 3:: TFM-GENERIC-NS-INIT-T-1

Index	**TFM-GENERIC-NS-INIT-T-1**
Description	The NS image can be tampered by an attacker
Justification	An attack may tamper the NS image to inject malicious code
Category	Tampering
Mitigation	By default TF-M relies on MCUBoot to validate NS image. The validation of NS image integrity and authenticity is completed in secure boot before jumping to NS entry or booting up NS core. Refer to [SECURE-BOOT] for more details. The validation may vary in diverse vendor platforms specific Chain of Trust (CoT) implementation.
CVSS Score	3.5 (Low)
CVSS Vector String	CVSS:3.1/AV:P/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N

Table 4:: TFM-GENERIC-NS-INIT-T-2

Index	**TFM-GENERIC-NS-INIT-T-2**
Description	An attacker may replace the current NS image with an older version.
Justification	The attacker downgrades the NS image with an older version which has been deprecated due to known security issues. The older version image can pass the image signature validation and its vulnerabilities can be exploited by attackers.
Category	Tampering
Mitigation	TF-M relies on MCUBoot to perform anti-rollback protection. TF-M defines a non-volatile counter API to support anti-rollback. Each platform must implement it using specific trusted hardware non-volatile counters. For more details, refer to [ROLLBACK-PROTECT]. The anti-rollback protection implementation can vary on diverse platforms.
CVSS Score	3.5 (Low)
CVSS Vector String	CVSS:3.1/AV:P/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N

Table 5:: TFM-GENERIC-NS-INIT-T-I-1

Index	**TFM-GENERIC-NS-INIT-T-I-1**
Description	If SPE doesn't complete isolation configuration before NSPE starts, NSPE can access secure regions which it is disallowed to.
Justification	Secure data can be tampered or disclosed if NSPE is activated and accesses secure regions before isolation configuration is completed by SPE.
Category	Tampering/Information disclosure
Mitigation	SPE must complete and enable proper isolation to protect secure regions from being accessed by NSPE, before jumping to NS entry or booting up NS core. TF-M executes isolation configuration at early stage of secure initialization before NS initialization starts. On dual-cpu platform, platform specific initialization must halt NS core until isolation is completed, as defined in [DUAL-CPU-BOOT]. TF-M defines isolation configuration HALs for platform implementation. The specific isolation configuration depends on platform specific implementation.
CVSS Score	9.0 (Critical)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N

Table 6:: TFM-GENERIC-NS-INIT-T-I-2

Index	**TFM-GENERIC-NS-INIT-T-I-2**
Description	If SPE doesn't complete isolation configuration before NSPE starts, NSPE can control devices or peripherals which it is disallowed to.
Justification	On some platforms, devices and peripherals can be configured as Secure state in runtime. If security status configuration of those device and peripherals are not properly completed before NSPE starts, NSPE can control those device and peripherals and may be able to tamper data or access secure data.
Category	Tampering/Information disclosure
Mitigation	SPE must complete and enable proper configuration and isolation to protect critical devices and peripherals from being accessed by NSPE, before jumping to NS entry or booting up NS core. TF-M executes isolation configuration of devices and peripherals at early stage of secure initialization before NS initialization starts. The specific isolation configuration depends on platform specific implementation.
CVSS Score	9.0 (Critical)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N

Table 7:: TFM-GENERIC-NS-INIT-I-2

Index	**TFM-GENERIC-NS-INIT-I-2**
Description	If SPE leaves some SPE information in non-secure memory or shared registers when NSPE starts, NSPE may access those SPE information.
Justification	If NSPE can access those SPE information from shared registers or non-secure memory, secure information may be disclosed.
Category	Information disclosure
Mitigation	SPE must clean up the secure information from shared registers before NS starts. TF-M invalidates registers not banked before handing over the system to NSPE on single Armv8-M platform. On dual-cpu platforms, shared registers are implementation defined, such as Inter-Processor Communication registers. Dual-cpu platforms must not store any data which may disclose secure information in the shared registers. SPE must avoid storing SPE information in non-secure memory.
CVSS Score	4.3 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:L/I:N/A:N

Table 8:: TFM-GENERIC-NS-INIT-D-1

Index	**TFM-GENERIC-NS-INIT-D-1**
Description	An attacker may block NS to boot up
Justification	An attacker may block NS to boot up, such as by corrupting NS image, to stop the whole system from performing normal functionalities.
Category	Denial of service
Mitigation	No SPE information will be disclosed and TF-M won't be directly impacted. It relies on NSPE and platform specific implementation to mitigate this threat. It is out of scope of this threat model.
CVSS Score	4.0 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L

NSPE requests TF-M secure service

This section identifies threats on DF2 defined in *Data Flow Diagram*.

Table 9:: TFM-GENERIC-REQUEST-SERVICE-S-1

Index	**TFM-GENERIC-REQUEST-SERVICE-S-1**
Description	A malicious NS application may pretend as a secure client to access secure data which NSPE must not directly access.
Justification	[FF-M] defines `Client ID` to distinguish clients which request RoT services. Secure clients are assigned with positive IDs and non-secure clients are assigned with negative ones. A malicious NS application may provide a positive `Client ID` to pretend as a secure client to access secure data.
Category	Spoofing
Mitigation	TF-M checks the `Client ID` from NSPE. If the NS `Client ID` is not a valid one, TF-M will report this as a security error.
CVSS Score	8.4 (High)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:N

Table 10:: TFM-GENERIC-REQUEST-SERVICE-T-1

Index	**TFM-GENERIC-REQUEST-SERVICE-T-1**
Description	An attacker in NSPE may tamper the service request input or output vectors between check and use (Time-Of-Check-to-Time-Of-Use (TOCTOU)).
Justification	If SPE validates the content in input/output vectors locally in NSPE memory, an attacker in NSPE can have a chance to tamper the content after the validation successfully passes. Then SPE will provide RoT service according to the corrupted parameters and it may cause further security issues.
Category	Tampering
Mitigation	In TF-M implementation, the validation of NS input/output vectors are only executed after those vectors are copied from NSPE into SPE. It prevents an attack from NSPE to tamper those parameters after validation in TF-M.
CVSS Score	7.8 (High)
CVSS Vector String	CVSS:3.1/AV:L/AC:H/PR:N/UI:N/S:C/C:H/I:H/A:N

Table 11:: TFM-GENERIC-REQUEST-SERVICE-T-2

Index	**TFM-GENERIC-REQUEST-SERVICE-T-2**
Description	A malicious NS application may request to tamper data belonging to SPE.
Justification	A malicious NS application may request SPE RoT services to write malicious value to SPE data. The malicious NS application may try to tamper SPE assets, such as keys, or modify configurations in SPE. The SPE data belongs to components in SPE and must not be accessed by NSPE.
Category	Tampering
Mitigation	TF-M executes memory access check to all the RoT service requests. If a request doesn't have enough permission to access the target memory region, TF-M will refuse this request and assert a security error.
CVSS Score	7.1 (High)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:N/I:H/A:N

Table 12:: TFM-GENERIC-REQUEST-SERVICE-R-1

Index	**TFM-GENERIC-REQUEST-SERVICE-R-1**
Description	A NS application may repudiate that it has requested services from a RoT service.
Justification	A malicious NS application may call a RoT service to access critical data in SPE, which it is disallowed to, via a non-public vulnerability. It may refuse to admit that it has accessed that data.
Category	Repudiation
Mitigation	TF-M implements an event logging secure service to record the critical events, such as the access to critical data.
CVSS Score	0.0 (None)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:N

Table 13:: TFM-GENERIC-REQUEST-SERVICE-I-1

Index	**TFM-GENERIC-REQUEST-SERVICE-I-1**
Description	A malicious NS application may request to read data belonging to SPE.
Justification	A malicious NS application may request SPE RoT services to copy SPE data to NS memory. The SPE data belongs to components in SPE and must not be disclosed to NSPE, such as root keys.
Category	Information disclosure
Mitigation	TF-M executes memory access check to all the RoT service requests. If a request doesn't have enough permission to access the target memory region, TF-M will refuse this request and assert a security error.
CVSS Score	7.1 (High)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:N/A:N

Table 14:: TFM-GENERIC-REQUEST-SERVICE-T-I-1

Index	**TFM-GENERIC-REQUEST-SERVICE-T-I-1**
Description	A malicious NS application may request to control secure device and peripherals, on which it doesn't have the permission.
Justification	A malicious NS application may request RoT services to control secure device and peripherals, on which it doesn't have the permission.
Category	Tampering/Information disclose
Mitigation	TF-M performs client check to validate whether the client has the permission to access the secure device and peripherals.
CVSS Score	9.0 (Critical)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N

Table 15:: TFM-GENERIC-REQUEST-SERVICE-D-1

Index	**TFM-GENERIC-REQUEST-SERVICE-D-1**
Description	A Malicious NS applications may frequently call secure services to block secure service requests from other NS applications.
Justification	TF-M runs on IoT devices with constrained resource. Even though multiple outstanding NS PSA Client calls can be supported in system, the number of NS PSA client calls served by TF-M simultaneously are still limited. Therefore, if a malicious NS application or multiple malicious NS applications continue calling TF-M secure services frequently, it may block other NS applications to request secure service from TF-M.
Category	Denial of service
Mitigation	TF-M is unable to manage behavior of NS applications. Assets are not disclosed and TF-M is neither directly impacted in this threat. It relies on NS OS to enhance scheduling policy and prevent a single NS application to occupy entire CPU time. It is beyond the scope of this threat model.
CVSS Score	4.0 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L

Table 16:: TFM-GENERIC-REQUEST-SERVICE-D-2

Index	**TFM-GENERIC-REQUEST-SERVICE-D-2**
Description	A malicious NS application may provide invalid NS memory addresses as the addresses of input and output data in RoT service requests.
Justification	SPE may be unable to achieve full knowledge of NS memory mapping. SPE may fail to capture those invalid NS memory addresses during memory access check since those invalid addresses may not be included in isolation configuration. In that case, SPE will access those invalid NS memory addresses later to read or write data. It may trigger a system error to crash the whole system immediately. The malicious NS application may be blocked by NS MPU from directly accessing that invalid NS memory address. But it may manipulate SPE to access that address instead.
Category	Denial of service
Mitigation	TF-M executes memory access check to the memory addresses in all the NS requests. On single Armv8-M core platforms, TF-M invokes `TT` instructions to execute memory address check. If a NS memory area is not matched in any valid SAU or MPU region, it will be marked as invalid and any access permission is disallowed. Therefore, SPM will reject any NS request containing invalid NS memory addresses and reports it as as a security error. On dual-core platforms, TF-M implements a default memory access check. If a NS memory area is not found in any memory region configured for isolation, it will be marked as invalid and therefore SPM will reject the corresponding NS request. It will be reported as a security error. Dual-core platforms may implement platform specific memory check to replace the default one. It relies on platform specific implementation to capture invalid memory address. It is out of the scope of this document.
CVSS Score	3.2 (Low)
CVSS Vector String	CVSS:3.1/AV:L/AC:H/PR:N/UI:N/S:C/C:N/I:N/A:L

RoT services read and write NS data

This section identifies threats on DF3 defined in *Data Flow Diagram*.

RoT services can either directly access NS memory or rely on TF-M SPM to obtain NS input data and send response data back to NS memory.

Table 17:: TFM-GENERIC-SECURE-SERVICE-RW-T-1

Index	**TFM-GENERIC-SECURE-SERVICE-RW-T-1**
Description	An attacker may tamper NS input data while the RoT service is processing those data.
Justification	A RoT service may access NS input data multiple times during its data processing. For example, it may validate or authenticate the NS input data before it performs further processing. If the NS input data remains in NSPE memory during the RoT service execution, an attacker may tamper the NS input data in NSPE memory after the validation passes.
Category	Tampering
Mitigation	If RoT services request SPM to read and write NS data. TF-M SPM follows [FF-M] to copy the NS input data into SPE memory region owned by the RoT service, before the RoT service processes the data. Therefore, the NS input data is protected during the RoT service execution from being tampered. If RoT services can directly access NS memory and read NS input data multiple times during data processing, it is required to review and confirm the implementation of the RoT service copies NS input data into SPE memory area before it processes the data.
CVSS Score	3.2 (Low)
CVSS Vector String	CVSS:3.1/AV:L/AC:H/PR:N/UI:N/S:C/C:N/I:L/A:N

Table 18:: TFM-GENERIC-SECURE-SERVICE-RW-T-2

Index	**TFM-GENERIC-SECURE-SERVICE-RW-T-2**
Description	A malicious NS application may embed secure memory addresses into a structure in RoT service request input vectors, to tamper secure memory which the NS application must not access.
Justification	[FF-M] limits the total number of input/output vectors to 4. If a RoT service requires more input/output vectors, it may define a parameter structure which embeds multiple input/output buffers addresses. However, as a potential security risk, a malicious NS application can put secure memory addresses into a valid parameter structure to bypass TF-M validation on those memory addresses. The parameter structure can pass TF-M memory access check since itself is valid. However, if the RoT service parses the structure and directly write malicious data from NSPE to the secure memory addresses in parameter structure, the secure data will be tampered.
Category	Tampering
Mitigation	It should be avoided to embed memory addresses into a single input/output vector. If more than 4 memory addresses are required in a RoT service request, it is recommended to split this request into two or multiple service calls and therefore each service call requires no more than 4 input/output vectors. If RoT services request SPM to read and write NS data. SPM will validate the target addresses and can detect the invalid addresses to mitigate this threat. If RoT services can directly access NS memory, it is required to review and confirm the implementation of RoT service request doesn't embed memory addresses.
CVSS Score	7.1 (High)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:N/I:H/A:N

Table 19:: TFM-GENERIC-SECURE-SERVICE-RW-I-1

Index	**TFM-GENERIC-SECURE-SERVICE-RW-I-1**
Description	Similar to *TFM-GENERIC-SECURE-SERVICE-RW-T-2*, a malicious NS application can embed secure memory addresses in a parameter structure in RoT service request input vectors, to read secure data which the NS application must not access.
Justification	Similar to the description in *TFM-GENERIC-SECURE-SERVICE-RW-T-2*, the secure memory addresses hidden in the RoT service input/output vector structure may bypass TF-M validation. Without a proper check, the RoT service may copy secure data to NSPE according to the secure memory addresses in structure, secure information can be disclosed.
Category	Information disclosure
Mitigation	It should be avoided to embed memory addresses into a single input/output vector. If more than 4 memory addresses are required in a RoT service request, it is recommended to split this request into two or multiple service calls and therefore each service call requires no more than 4 input/output vectors. If RoT services request SPM to read and write NS data. SPM will validate the target addresses and can detect the invalid addresses to mitigate this threat. If RoT services can directly access NS memory, it is required to review and confirm the implementation of RoT service request doesn't embed memory addresses.
CVSS Score	7.1 (High)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:N/A:N

TF-M returns secure service result

This section identifies threats on `DF4` defined in *Data Flow Diagram*.

When RoT service completes the request from NSPE, TF-M returns the success or failure error code to NS application.

In single Armv8-M core scenario, TF-M writes the return code value in the general purpose register and returns to Non-secure state.

On dual-cpu platforms, TF-M writes the return code to NSPE mailbox message queue via mailbox.

Table 20:: TFM-GENERIC-RETURN-CODE-I-1

Index	**TFM-GENERIC-RETURN-CODE-I-1**
Description	SPE may leave secure data in the registers not banked after the SPE completes PSA Client calls and executes `BXNS` to switch Armv8-M back to Non-secure state.
Justification	If SPE doesn't clean up the secure data in registers not banked before switching into NSPE in Armv8-M core, NSPE can read the SPE context from those registers.
Category	Information disclosure
Mitigation	In single Armv8-M core scenario, TF-M cleans general purpose registers not banked before switching into NSPE to prevent NSPE probing secure context from the registers. When FPU is enabled in TF-M, secure FP context belonging to a secure partition will be saved on this partition's stack and cleaned by hardware during context switching. Also TF-M cleans secure FP context in FP registers before switching into NSPE to prevent NSPE from probing secure FP context.
CVSS Score	4.3 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:L/I:N/A:N

NS interrupts preempts SPE execution

This section identifies threats on DF5 defined in *Data Flow Diagram*.

Table 21:: TFM-GENERIC-NS-INTERRUPT-I-1

Index	**TFM-GENERIC-NS-INTERRUPT-I-1**
Description	Shared registers may contain secure data when NS interrupts occur.
Justification	The secure data in shared registers should be cleaned up before NSPE can access shared registers. Otherwise, secure data leakage may occur.
Category	Information disclosure
Mitigation	In single Armv8-M core scenario, Armv8-M architecture automatically cleans up the registers not banked before switching to Non-secure state while taking NS interrupts. When FPU is enabled in TF-M, with setting of FPCCR_S.TS = 1 besides secure FP context in FP caller registers, FP context in FP callee registers will also be cleaned by hardware automatically when NS interrupts occur, to prevent NSPE from probing secure FP context in FP registers. Refer to Armv8-M Architecture Reference Manual[ARM arm]_ for details. On dual-cpu platforms, shared registers are implementation defined, such as Inter-Processor Communication registers. Dual-cpu platforms must not store any data which may disclose secure information in the shared registers.
CVSS Score	4.3 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:L/I:N/A:N

Table 22:: TFM-GENERIC-NS-INTERRUPT-D-1

Index	**TFM-GENERIC-NS-INTERRUPT-D-1**
Description	An attacker may trigger spurious NS interrupts frequently to block SPE execution.
Justification	In single Armv8-M core scenario, an attacker may inject a malicious NS application or hijack a NS hardware to frequently trigger spurious NS interrupts to keep preempting SPE and block SPE to perform normal secure execution.
Category	Denial of service
Mitigation	It is out of scope of TF-M. Assets protected by TF-M won’t be leaked. TF-M won’t be directly impacted.
CVSS Score	4.0 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L

Secure interrupts preempts NSPE execution

This section identifies threats on DF6 defined in *Data Flow Diagram*.

Table 23:: TFM-GENERIC-S-INTERRUPT-I-1

Index	**TFM-GENERIC-S-INTERRUPT-I-1**
Description	Shared registers may contain secure data when Armv8-M core switches back to Non-secure state on Secure interrupt return.
Justification	Armv8-M architecture doesn't automatically clean up shared registers while returning to Non-secure state during Secure interrupt return. If SPE leaves critical data in the Armv8-M registers not banked, NSPE can read secure context from those registers and secure data leakage may occur.
Category	Information disclosure
Mitigation	TF-M saves NPSE context in general purpose register R4~R11 into secure stack during secure interrupt entry. After secure interrupt handling completes, TF-M unstacks NSPE context from secure stack to overwrite secure context in R4~R11 before secure interrupt return. Armv8-M architecture will automatically unstack NSPE context from non-secure stack to overwrite other registers not banked, such as R0~R3 and R12, during secure interrupt return, before NSPE software can access those registers. When FPU is enabled in TF-M, with setting of FPCCR_S.TS = 1 and FPCCR_S.CLRONRET = 1, besides secure FP context in FP caller registers, FP context in callee registers will also be cleaned by hardware automatically during S exception return, to prevent NSPE from probing secure FP context in FP registers. Refer to Armv8-M Architecture Reference Manual [ARM arm]_ for details.
CVSS Score	4.3 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:L/I:N/A:N

Miscellaneous threats

This section collects threats irrelevant to the valid TF-M data flows shown above.

Table 24:: TFM-GENERIC-STACK-SEAL

Index	**TFM-GENERIC-STACK_SEAL**
Description	Armv8-M processor Secure software Stack Sealing vulnerability.
Justification	On Armv8-M based processors with TrustZone, if Secure software does not properly manage the Secure stacks when the stacks are created, or when performing non-standard transitioning between states or modes, for example, creating a fake exception return stack frame to de-privilege an interrupt, it is possible for Non-secure world software to manipulate the Secure Stacks, and potentially influence Secure control flow. Refer to [STACK-SEAL] for details.
Category	Elevation of privilege
Mitigation	TF-M has implemented common mitigation against stack seal vulnerability. Refer to [ADVISORY-TFMV-1] for details on analysis and mitigation in TF-M.
CVSS Score	5.3 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:L/I:L/A:L

Table 25:: TFM-GENERIC-SVC-CALL-SP-FETCH

Index	**TFM-GENERIC-SVC-CALL-SP-FETCH**
Description	Invoking Secure functions from handler mode may cause TF-M IPC model to behave unexpectedly.
Justification	On Armv8-M based processors with TrustZone, if NSPE calls a secure function via Secure Gateway (SG) from non-secure Handler mode , TF-M selects secure process stack by mistake for SVC handling. It will most likely trigger a crash in secure world or reset the whole system, with a very low likelihood of overwriting some memory contents.
Category	Denial of service/Tampering
Mitigation	TF-M has enhanced implementation to mitigate this vulnerability. Refer to [ADVISORY-TFMV-2] for details on analysis and mitigation in TF-M.
CVSS Score	4.5 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:H/PR:N/UI:N/S:C/C:N/I:L/A:L

Table 26:: VLLDM instruction security vulnerability

Index	**TFM-GENERIC-FP-VLLDM**
Description	Secure data in FP registers may be disclosed to NSPE when VLLDM instruction is abandoned due to an exception mid-way.
Justification	Refer to [VLLDM Vulnerability]_ for details.
Category	Tampering/Information disclosure
Mitigation	In current TF-M implementation, when FPU is enabled in SPE, TF-M configures NSACR to disable NSPE to access FPU. Therefore, secure data in FP registers is protected from NSPE. Refer to [VLLDM Vulnerability]_, for details on analysis and mitigation.
CVSS Score	3.4 (Low)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:U/C:L/I:L/A:N

3.1.6 Version control

Table 27:: Version control

Version	Description	TF-M version
v0.1	Initial draft	TF-M v1.1
v1.0	First version	TF-M v1.2.0
v1.1	Update version	TF-M v1.5.0
v1.2	Update details to align FP support in NSPE.	TF-M v1.5.0

3.1.7 Reference

3.2 Security Advisories

3.2.1 Advisory TFMV-1

Title	NS world may cause the CPU to perform an unexpected return operation due to unsealed stacks.
CVE ID	CVE-2020-16273
Date	16 October 2020
Versions Affected	All versions up to and including TF-M v1.1
Configurations	All
Impact	Most likely a crash in secure world with a very low likelihood of hijacking secure world execution flow via a separate vulnerability.
Fix Version	commit 92e46a3abd0e328fac29ccd1cf161cd482397567
Credit	Matvey Mukha

Background

When the Non-Secure world returns to Secure after a callback (FNC_RETURN) or Non-Secure interrupt handling (EXC_RETURN), the hardware pops the secure return address and RET_PSR from the respective secure stack. The hardware performs a set of integrity checks at this point and will raise a fault if the checks fail. There is a potential vulnerability if the NS attempts a wrong FNC_RETURN or EXC_RETURN and causes the PE to pop from the unexpected stack. Please refer to ARMv8-M Secure stack sealing advisory notice for more details on the vulnerability.

To prevent such an attack, the architecture expects the secure software to *seal* the stacks. The ARMv8-M ARM states that the stack should be sealed with the special value, 0xFEF5EDA5 (informative IGJGL).

Both the MSP_S and the PSP_S stacks need to be sealed to mitigate stack underflow attacks and this is not done currently in TF-M.

Impact

This section analyses the impact of this vulnerability on the upstream TF-M implementation.

All requests coming in from the Non-Secure world uses ARM_LIB_STACK as the PSP_S and then switches to MSP_S as part of SPM scheduling. The MSP_S is fully unwound when the scheduled partition is executing. All partitions run using the PSP_S and this stack can be separate for each partition or be a common stack depending on whether TF-M is in library mode or IPC model. When the partition execution using PSP_S switches to non-secure world due to a non-secure interrupt or a non-secure callback invocation, the non-secure world on return back to secure can use a fake EXC_RETURN or FNC_RETURN operation to trigger an MSP_S stack underflow, and the CPU could fetch the return PC and PSR from the memory just above MSP_S stack (stack grows from higher to lower address).

The memory above MSP_S is the ARM_LIB_STACK (as described by tfm_common_s.sct/ld), which because of overlaying or zero initialization is most likely to be initialized memory for most platforms. This is the stack used to run the initialization code of TF-M and usually there will some unused free space in the stack. Any underflow of MSP_S will be into this stack and hence is likely to be a deterministic crash given that this memory is initialized. In theory, a separate vulnerability that allows an attacker to control the memory above MSP_S in concert with this vulnerability could enable an attacker to hijack the secure world execution flow but the likelihood of that is very low.

Through analysing TF-M specific initialization and execution flow, we have not found any scenario in TF-M in which Non-secure world can fake a return operation back to secure world and cause underflow of PSP_S.

As described in the white paper, de-privileged interrupt handling is also vulnerable to this problem. The Library mode of TF-M uses de-privileged interrupt handling and does not allow non-secure interrupt to pre-empt the secure interrupt handling. But if the de-privileged handler makes a Non-Secure callback then there is a chance that Non-Secure world could return back to Secure world via a fake FNC_RETURN. Under certain conditions, this can force the CPU to use the 2 words on top of stack as return PC and PSR. Sealing the top of MSP_S before switching to PSP_S as part of de-privileged interrupt handling mitigates this vulnerability.

The interrupt handling in IPC model uses PSA signal to signal the partition and does not use de-privileged interrupt handling mechanism. The PSA signal gets handled by the partition when it is scheduled to run by the SPM. Hence during interrupt handling in IPC model, there is no additional threat caused by this vulnerability, compared to regular partition execution.

Conclusion

Overall, from analysis of upstream TF-M implementation, the severity of this vulnerability is low, given that the values popped out from secure stack (either via underflow of MSP_S or from top of MSP_S stack in de-privileged interrupt handling) cannot be influenced by the Non Secure world. To mitigate the risk completely, the fix in TF-M is to follow the recommendation in ARMv8-M ARM which is to seal the base of both handler mode stack (MSP_S) and thread stacks (PSP_S) and, in case of library mode, also seal the top of MSP_S before handing control over to de-privileged interrupt handler.

3.2.2 Advisory TFMV-2

Title	Invoking Secure functions from handler mode may cause TF-M IPC model to behave unexpectedly.
CVE ID	CVE-2021-27562
Date	Mar 5, 2021
Versions Affected	Affected All versions up to and including TF-M v1.2
Configurations	IPC Model on Armv8-M
Impact	Most likely a crash in secure world or reset whole system, with a very low likelihood of overwriting some memory contents.
Fix Version	commit e212ea1637a66255b44d0e7c19ebe9786ab56ccb
Credit	Øyvind Rønningstad, Senior SW Engineer from Nordic Semiconductor

Background

When a non-secure caller calls the secure function, the execution switches the security state via Secure Gateway (SG), then arrived at the TF-M secure entry if the SG is successful. After SG, TF-M invokes SVC to SPM at first because SPM handles requests from SPE and NSPE in a unified SVC handler. The TF-M SVC handler code relies on the 'SPSEL' bit in 'EXC_RETURN' to get the caller stack pointer:

```
; Armv8m mainline as the example
mrs         r2, msp                 ; r2 = msp;
tst         lr, #4                  ; EXC_RETURN.SPSEL == ?
ite         eq                      ; condition preparation
```

(continues on next page)

(continued from previous page)

```
moveq     r0, r2                ; if (EXC_RETURN.SPSEL == 0) r0 = msp;
movne     r0, psp               ; if (EXC_RETURN.SPSEL == 1) r0 = psp;
mov       r1, lr                ; r1 = EXC_RETURN;
bl        tfm_core_svc_handler ; r0 = sp(context), r1 = EXC_RETURN, r2 = msp
```

If NSPE calls the secure function in 'Handler mode', the TF-M secure entry runs in 'Handler mode' before invoking SVC because PE mode does not change during the transition from non-secure state to secure state via the successful SG. Main stack pointer (MSP) is always used in 'Handler mode', which is irrelevant to the value of SPSEL. However, the code above selects secure process stack pointer (PSP_S, S suffix here indicates the security state) for further SVC handling based on EXC_RETURN.SPSEL, hence the SVC handler accesses the wrong stack for handling the request in the NSPE caller in 'Handler mode' case. To prevent this vulnerability, TF-M secure SVC handler should fetch the right stack pointer register by checking both the PE mode and SPSEL bit. The handler should also prevent callers in non-secure *Handler mode* from calling TF-M SVC functionalities. The following section analysis impact of the IPC model. Library model is not vulnerable to this attack because it checks the PE mode in the TF-M secure entry.

Impact

When the vulnerability is happening, PSP_S is the incorrect stack pointer fetched, the impact is decided by the content PSP_S is pointing to. The SVC handler gets the SVC number by referencing the memory at the 2 bytes subtracted position of member 'Return Address' in the PSP_S pointing context, and uses the caller saved registers in the context as parameters for the subsequent functions indicated by the SVC number. The PSP_S may point to the bottom (higher address) of secure thread stack if there is no ongoing secure function call, or it points to a preempted context caused by non-secure preempting secure execution. When PSP_S is pointing to the stack bottom when this issue happens, the caller context is pointing to the underflowed stack of which the top 2 words are stack seal, and the rest of the context is unpredictable. These underflowed content are ZERO initialized for most platforms, hence when fetching the SVC Number at memory address 'Return Address - 2', a 'MemFault' or 'HardFault' would happen. Even if a valid SVC number is returned, the stack seal values are part of the parameters for the subsequent functions that have parameters, which cannot pass the validation for the parameters. When PSP_S is pointing to a preempted context, the preempted place can be the TF-M secure entry area or internal thread space. If the preemption happens when the execution is in the TF-M secure entry area, the PSP_S will point to an NS preemption stack frame and hence will fail the context size validation through this entry into TF-M SVC Handler. In the other scenario that TF-M internal thread is preempted, there is a very remote chance that the exception stack frame will contain a context with a valid SVC Number with valid parameters. As a summary, the impacts of the vulnerability depend on whether an SVC number could be fetched from the incorrect stack and what operation the fetched SVC number corresponds to:

- A memory access fault when fetching the SVC number from memory. This fault halts the whole system. A reset can recover the system back to normal.
- The SVC number corresponds to the initialization process in TF-M. SPM initialization is called for the second time. This re-initialization process will fail and halt the whole system. A reset can recover the system back to normal.
- The SVC number corresponds to the boot data fetching function. If the unpredictable data in stack triggers boot data fetching and passes the parameter validation under a very rare case, boot data is copied into an unpredictable memory address. One note here, the TF-M default boot data has no sensitive information.
- The SVC number corresponds to the log output function. If the unpredictable data in stack triggers log output under a very rare case, secure data at unpredictable address might be printed out through the log interface.
- The SVC number corresponds to other valid numbers other than above. The system resets because of parameter validation fail.

Conclusion

Overall, from the analysis of upstream TF-M implementation, the severity of this vulnerability is Medium, given that a software crash or system reset happen most of the time. The probability for causing secure data leakage via log output or tampering of secure memory with boot data is extremely low as the TF-M internal thread exception stack frame contents have to pass all the validations performed by SPM.

3.2.3 Advisory TFMV-3

Title	`abort()` function may not take effect in TF-M Crypto multi-part MAC/hashing/cipher operations.
CVE ID	CVE-2021-32032
Public Disclosure Date	May 10, 2021
Versions Affected	Affected all versions up to and including TF-M v1.3.0
Configurations	All
Impact	It can cause memory leakage in TF-M Crypto service, eventually making TF-M Crypto service unavailable and impacting other services relied on it.
Fix Version	commit 7e2e52
Credit	Chongqing Lei, Southeast University Zhen Ling, Associate Professor, Southeast University Xinwen Fu, Professor, University of Massachusetts Lowell

Background

PSA multi-part crypto operation sequence

PSA Crypto API specification defines a common sequence for all multi-part crypto operations. The sequence can be simplified to the following steps:

- `setup()` sets up the multi-part operation.
- `update()` adds data/configurations into the multi-part operation.
- `finish()` completes the multi-part operation.

PSA Crypto API specification requests that the corresponding `abort()` function shall be called when `update()` or `finish()` function fails. The `abort()` function aborts the ongoing multi-part operation and cleans up the operation context.

TF-M multi-part crypto operation functions eventually call the underlying crypto library (Mbed TLS by default) to perform those steps, including `abort()` step.

PSA multi-part crypto operation objects

PSA Crypto API specification defines an operation object for each type of multi-part crypto operations. For example, `psa_mac_operation_t` for multi-part MAC operations and `psa_hash_operation_t` for multi-part hashing operations.

TF-M Crypto service relies on the underlying crypto library (Mbed TLS by default) to implement those objects. The structures of those objects are crypto library specific and hidden to TF-M. The underlying crypto library usually stores and manages the context of ongoing multi-part crypto operations in the corresponding PSA operation object. For example, Mbed TLS stores multi-part hashing operation context in its `psa_hash_operation_t` implementation.

The context is cleaned up in crypto library `abort()` function when the client calls `abort()` to handle a previous error. The clean-up execution can include zeroing the memory area and freeing allocated memory.

TF-M multi-part crypto operation objects

TF-M Crypto service defines a dedicated operation structure `tfm_crypto_operation_s` to wrap PSA multi-part crypto operation object and maintains its own status, as shown in the code block below.

```
struct tfm_crypto_operation_s {

    ...

    union {
        psa_cipher_operation_t cipher;    /*!< Cipher operation context */
        psa_mac_operation_t mac;          /*!< MAC operation context */
        psa_hash_operation_t hash;        /*!< Hash operation context */
        psa_key_derivation_operation_t key_deriv; /*!< Key derivation operation context␣
↪*/
    } operation;
};
```

TF-M Crypto service assigns a `tfm_crypto_operation_s` object for each multi-part crypto operation sequence during `setup()` step. The `tfm_crypto_operation_s` object content will be cleaned after the sequence completes or fails.

Impact

During multi-part hashing/MAC/cipher operations, if the underlying crypto library function returns an error code, TF-M `update()` and `finish()` functions will immediately clean up the structure `tfm_crypto_operation_s` content and exit.

When `tfm_crypto_operation_s` content is cleaned in TF-M `update()` and `finish()` functions, the content in PSA multi-part crypto operation object inside `tfm_crypto_operation_s` is also cleaned. If the underlying crypto library stores operation context in the PSA operation object, the operation context is lost before clients call `abort()` to handle the error.

Therefore, the underlying crypto library `abort()` function can be unable to perform normal abort operation if it cannot fetch the context or its content. In other words, the underlying crypto library `abort()` may not work normally or take effect.

In theory when the case analyzed above occurs:

- If the underlying crypto library dynamically allocates some memory regions during multi-part operation and stores those memory region pointers in the PSA multi-part operation object, the underlying crypto library will be unable to locate and free those allocated memory regions in `abort()`. It will cause memory leakage in TF-M Crypto service. It may further make TF-M Crypto service unavailable and affect other services relying on TF-M Crypto service.

- The underlying crypto library `abort()` may still consider the field values in the context as valid. `abort()` may perform unexpected behaviors or access invalid memory regions. It may trigger further faults and block TF-M Crypto service or even the whole system.

Note: The actual consequences depend on the implementation of the multi-part operations in the underlying crypto library.

Impacted PSA Crypto API functions

The following PSA multi-part crypto operation functions are impacted:

- Multi-part hashing operations
  - `psa_hash_update()`
  - `psa_hash_finish()`
  - `psa_hash_verify()`
  - `psa_hash_clone()`
- Multi-part MAC operations
  - `psa_mac_update()`
  - `psa_mac_sign_finish()`
  - `psa_mac_verify_finish()`
- Multi-part cipher operations
  - `psa_cipher_generate_iv()`
  - `psa_cipher_set_iv()`
  - `psa_cipher_update()`
  - `psa_cipher_finish()`

Justifications on unaffected multi-part operations

TF-M multi-part AEAD operations and multi-part key derivation operations are not impacted by this issue.

TF-M Crypto service has not implemented multi-part AEAD operations. TF-M multi-part AEAD functions directly return an error of unsupported operations.

In TF-M key derivation implementation, the `psa_key_derivation_operation_t` object is only cleaned in the `abort()` function after the underlying crypto library completes abort.

Mitigation

The clean-up operation shall be removed from error handling routines in the following TF-M Crypto functions:

- Multi-part hashing operations
  - `tfm_crypto_hash_update()`
  - `tfm_crypto_hash_finish()`
  - `tfm_crypto_hash_verify()`
  - `tfm_crypto_hash_clone()`
- Multi-part MAC operations
  - `tfm_crypto_mac_update()`
  - `tfm_crypto_mac_sign_finish()`
  - `tfm_crypto_mac_verify_finish()`
- Multi-part cipher operations
  - `tfm_crypto_cipher_generate_iv()`
  - `tfm_crypto_cipher_set_iv()`
  - `tfm_crypto_cipher_update()`
  - `tfm_crypto_cipher_finish()`

Note: This mitigation assumes that client follows the sequence specified in PSA Crypto API specification to call `abort()` when an error occurs during multi-part crypto operations.

3.2.4 Advisory TFMV-4

Title	NSPE may access secure keys stored in TF-M Crypto service in Profile Small with Crypto key ID encoding disabled.
CVE ID	CVE-2021-40327
Public Disclosure Date	22nd Nov, 2021
Versions Affected	TF-M v1.4.0
Configurations	Profile Small
Impact	In Profile Small, secure keys stored in Crypto service can be leaked to NSPE if NSPE acquires secure key IDs.
Fix Version	Commit 42e77b and v1.4.1
Credit	N/A

Background

TF-M Profile Small disabled Crypto key ID encoding with key owner client ID in TF-M v1.4.0 release.

When the Crypto key is stored into TF-M Crypto service, the key ID is not encoded with the client ID of key owner in Profile Small in TF-M v1.4.0. Therefore, TF-M Crypto service is unable to distinguish or validate owners of keys in Profile Small. NSPE can access the keys belonging to SPE in Profile Small in some scenarios.

Details

In TF-M v1.4.0, TF-M Crypto service by default relies on two mechanisms to validate key owners in key management.

- TF-M Crypto service maintains a key handle array. When a key is stored in Crypto service, the key ID and the key owner client ID are stored in the array. When a caller requests to access a key, TF-M Crypto service validates the request by comparing the caller client ID with the stored key client ID.
- Mbed TLS stores a special structure encoded by key owner client ID and the key ID. When a caller requests to access a key, Mbed TLS validates the request by comparing the caller client ID with the key client ID stored in that structure.

Secure clients are not isolated from each other in Profile Small and it doesn't require to validate key owner client ID between secure clients. Therefore, in TF-M v1.4.0, Profile Small disabled both mechanisms above to optimize the key storage size. The key directly or indirectly stored via `psa_import_key()` is not encoded with key owner client ID.

However, it also disables the validation of NS client ID when a NS client accesses keys stored in TF-M Crypto. NS clients can call `psa_open_key()`/`psa_export_key()` to access secure clients' keys stored via `psa_import_key()`, if NS clients acquire the key ID of secure clients.

Impact

Only TF-M Profile Small is impacted. All the other configurations or Profiles are not affected.

Analysis of RoT services in Profile Small

TF-M Profile Small enables Internal Trusted Storage (ITS), Crypto and Initial Attestation by default. The following analysis focuses on the impact on RoT services in Profile Small.

- ITS service doesn't create or store its own key in Crypto service. It is not impacted directly.
- Crypto service key derivation may be impacted.
  - `psa_key_derivation_output_key()` eventually stores the derived key in Crypto service. The stored derived keys can be accessed by a NS client if the NS client acquires the derived key ID value.
  - Platform specific implementation may store Hardware Unique Key (HUK) into Crypto service for key derivation from HUK via `psa_import_key()`.
    - Platform driver may import HUK as a temporary key into Crypto service during derivation and close the temporary key when derivation completes.

      If a NS client preempts the derivation and calls PSA Cryptography API to access temporary HUK data stored in Crypto service, the access will be captured by TF-M re-entry detection and rejected by TF-M SPE.
    - Platform driver may permanently store HUK via Crypto service for derivation and the key is still managed by Crypto service when NSPE is running.

      NS client can access HUK data via PSA Cryptography API if it acquires the key ID of stored HUK.

- Symmetric key algorithm based Initial Attestation temporarily stores symmetric Initial Attestation Key (IAK) in Crypto service during Initial Attestation Token generation. It imports symmetric IAK into Crypto service during generation and removes it from Crypto service when generation completes.

  If a NS client preempts the generation and calls PSA Cryptography API to access the temporary IAK data stored in Crypto service, the access will be captured by TF-M re-entry detection and rejected by TF-M SPE.

  Therefore, Initial Attestation is not impacted directly.

Impact on Profile Small default implementation

Default Profile Small RoT services don't initially call Crypto key derivation or store any secure key into Crypto service.

According to the analysis of RoT services above, device HUK can be accessed by NS clients and leaked to NSPE, in Profile Small default implementation, when all the following conditions are met.

- Platform specific implementation stores HUK in Crypto service, initially or during a derivation requested by NS client.
- HUK is still stored in Crypto service when NSPE is running.
- An NS client acquires the key ID of HUK in Crypto service and accesses HUK key via PSA Cryptography API.

Other vulnerabilities are not found yet so far.

Impact on vendor RoT services

If a vendor RoT service is integrated in Profile Small, its keys stored via `psa_import_key()` or derived from `psa_key_derivation_output_key()` can be accessed by NS client and leaked to NSPE when both following conditions are met.

- The secure key is stored in Crypto service when NSPE is running.
- An NS client acquires the key ID and accesses the key via PSA Cryptography API.

How NS client can acquire secure key ID is related to key management implementation of the underlying crypto library in TF-M Crypto service. With default Mbed TLS, NS hackers can import a NS key at first to obtain the rough base value of Mbed TLS key slots and then try a smaller subset of key ID values by brute-force.

Mitigation

This issue has been fixed by enforcing Mbed TLS key ID encoding with key owner client ID to be enabled.

This patch intended to optimize TF-M Crypto service key handle array and coincidentally fixed the issue.

v1.4.1 fixed this issue as a patch release.

3.2.5 Advisory TFMV-5

Title	psa_fwu_write() may cause buffer overflow in SPE.
CVE ID	CVE-2021-43619
Public Disclosure Date	Feb 11, 2022
Versions Affected	From 3e7129f to 921d0ea
Configurations	IPC model with Firmware Update partition enabled
Impact	In IPC model, the caller of psa_fwu_write() from SPE or NSPE can overwrite the stack memory outside of the local buffer in Firmware Update partition.
Fix Version	commit 78f7530
Credit	Mark Horvath, Staff Software Engineer from Arm Ltd.

Background

In Firmware Update partition, the psa_fwu_write() service is declared as:

```
/**
 * \brief Writes an image to its staging area.
 *
 * Writes the image data 'block' with length 'block_size' to its staging area.
 *
 * \param[in] image_id        The identifier of the image
 * \param[in] block_offset    The offset of the block being passed into block,
 *                            in bytes
 * \param[in] block           A buffer containing a block of image data. This
 *                            might be a complete image or a subset.
 * \param[in] block_size      Size of block. The size must not be greater than
 *                            PSA_FWU_MAX_BLOCK_SIZE.
 *
 */
psa_status_t psa_fwu_write(psa_image_id_t image_id,
                           size_t block_offset,
                           const void *block,
                           size_t block_size);
```

In IPC model, this service calls the tfm_fwu_write_ipc() API to write the input data into the device. In this API, the block_size bytes input data (in block argument) is read into a 1024 bytes local buffer via the psa_read() API. If the input argument block_size is greater than 1024, then the memory space starting from the address of the local buffer with block_size bytes would be overwritten by the input data in block argument.

Impact

In IPC model, the caller of `psa_fwu_write()` from SPE or NSPE can overwrite the memory space in RAM. The overwritten memory space ranges from the address of the local buffer which locates at the stack of Firmware Update partition to the end of the RAM. The overwritten memory may include the data of SPM and device drivers, as well as part of the stack of Firmware Update partition.

Mitigation

Add check against the input buffer length in the `tfm_fwu_write_ipc()` API before reading the input data into the local buffer. See commit 78f7530.

3.2.6 Advisory TFMV-6

Title	Partial tag comparison when using Chacha20-Poly1305 on the PSA driver API interface in CryptoCell enabled platforms
CVE ID	CVE-2023-40271
Public Disclosure Date	04/09/2023
Versions Affected	TF-M v1.6.0, TF-M v1.6.1, TF-M v1.7.0, TF-M v1.8.0
Configurations	CC312 enabled platforms, where the legacy driver API is disabled (`CC312_LEGACY_DRIVER_API_ENABLED=OFF`) and the single part AEAD APIs are implemented through a dedicated function and not by leveraging the multipart functions (`CC3XX_CONFIG_ENABLE_AEAD_ONE_SHOT_USE_MULTIPART` not set)
Impact	It might allow for unauthenticated payloads to be deemed as authentic by comparing only the first 4 bytes of the authentication tag instead of the full length of 16 bytes
Fix Version	2e82124af, TF-M v1.8.1
Credit	Nordic Semiconductor

Background

AEAD algorithms

Authenticated Encryption with Associated Data (AEAD) is a common ciphering method where the data to be encrypted is also authenticated as part of the process by creating an authentication tag. When the encrypted data is then decrypted, the authentication tag is verified and if it does not match the expected value, then the entire operation fails. In this way, the operation allows for Authenticity in addition to the confidentiality granted by the encryption process. Using PSA Crypto APIs, it's possible to use several of such algorithms, such as AES in Galois-Counter Mode (GCM), AES in

Counter with CBC-Mac Mode (CCM) or Chacha20-Poly1305, which is a combination of the Chacha20 cipher with the Poly1305 authenticator[1].

In particular for Chacha20-Poly1305 the corresponding macro defining the algorithm in PSA Cryptographic API specification is PSA_ALG_CHACHA20_POLY1305.

Single part vs multipart API functions

PSA Crypto API specification[2] allows the usage of AEAD algorithms through several possible APIs, that can be grouped generally in the single part, or integrated, operation type, and in the multipart operation type. The main difference is that for single part operations, the whole encryption and tag production (and on the other hand, the whole decryption and tag authentication) happen with a single API call. This type of approach is simpler but at the same time less flexible than the multipart approach where an operation context must be allocated by the application and the encryption/decryption processes require the call to different APIs to setup the operation context, provide inputs to the process and calculate the output. For example, the integrated APIs defined by the PSA Crypto API spec are called `psa_aead_encrypt()` and `psa_aead_decrypt`.

It is possible, to reduce the code size of an implementation, to implement the single part APIs by calling the underlying multipart functions, effectively encapsulating the multipart flow in the single part APIs.

PSA Unified Driver API for Cryptoprocessors

The PSA Unified Driver API for Cryptoprocessors spec[3] allows a PSA compliant implementation to redirect some of the operations to the underlying hardware accelerated platform. For example, CryptoCell is a cryptographic accelerator available in some of the TF-M supported platforms that can accelerate crypto operations, such as Chacha20-Poly1305. The driver code associated to it has the responsibility of driving the hardware resources with inputs and collect the outputs. When decrypting, the driver must compare the reconstructed authentication tag with the expected value, and return failure in case of mismatch during verification. For Chacha20-Poly1305, the tag size is 16 bytes.

For CryptoCell enabled platforms, the software component implementing the PSA Unified Driver interface is located into the `psa_driver_api` directory in `lib/ext/cryptocell-312-runtime/codesafe/src/` in TF-M's codebase. This components can be configured to have single part APIs implemented though the corresponding multipart functions by setting the following define at build time: `CC3XX_CONFIG_ENABLE_AEAD_ONE_SHOT_USE_MULTIPART`.

By default, CryptoCell enabled platforms don't build the PSA Unified Driver API interface layer but rely on the legacy interface. To enable the PSA Driver interface the following TF-M build option during CMake config must be be set to OFF: `CC312_LEGACY_DRIVER_API_ENABLED=OFF`.

Impact

When the PSA Driver API interface is used and is not configured to rely on the corresponding multipart functions, when performing the verification of the authentication tag at the end of the authenticated decryption process, the buffer containing the tag is only partially verified (the first 4 bytes only instead of the full 16 bytes). This allows for the possibility of unauthenticated data to be recognized as authentic. An attacker could theoretically construct a malicious payload to actively exploit such partial verification.

[1] Chacha20 and Poly1305 for IETF Protocols: https://datatracker.ietf.org/doc/html/rfc7539

[2] PSA Cryptographic API v1.1: https://armmbed.github.io/mbed-crypto/html/

[3] PSA Unified Driver interface: https://github.com/Mbed-TLS/mbedtls/blob/development/docs/proposed/psa-driver-interface.md

Impacted PSA Crypto API functions

The following PSA single part crypto operation function is impacted:

- `psa_aead_decrypt`

Mitigation

The verification of the authentication tag must happen on the full 16 bytes of instead of just the first 4 bytes. This means that loop that currently performs such verification in the `cc3xx_decrypt_chacha20_poly1305()` function must be changed from this:

```
/* Check tag in "constant-time" */
for (diff = 0, i = 0; i < sizeof(tag_length); i++)
    diff |= tag[i] ^ local_tag_buffer[i];
```

to this:

```
/* Check tag in "constant-time" */
for (diff = 0, i = 0; i < tag_length; i++)
    diff |= tag[i] ^ local_tag_buffer[i];
```

References

3.2.7 Advisory TFMV-7

Title	ARoT can access PRoT data via debug logging functionality
CVE ID	CVE-2023-51712
Public Disclosure Date	The issue was publicly reported on 2023.12.04
Versions Affected	All version up to TF-M v2.0.0 inclusive
Configurations	IPC mode with TFM_SP_LOG_RAW_ENABLED=1
Impact	A malicious ARoT partition can expose any part of memory via stdio interface if TFM_SP_LOG_RAW_ENABLED is set
Fix Version	TBD
Credit	Roman Mazurak, Infineon

Background

TF-M log subsystem if enabled by `TFM_SP_LOG_RAW_ENABLED` config option, uses a SVC call to print logging messages on the stdio output interface. Since the SVC handler has the highest privilege level and full memory access, this communication channel can be exploited to expose any memory content to stdout device, usually UART. The logging subsystem is available to the secure side only but in isolation level 2 and higher PSA Root of Trust partitions (PRoT) shall be protected from an access from Application Root of Trust (ARoT) partitions. Although a direct call of `tfm_hal_output_sp_log()` from ARoT partition will be blocked by MPU raising the `MemoryManagement()` exception, a malicious ARoT partition can create an alternative SVC call to output any memory data like this:

```
static int tfm_output_unpriv_string(const unsigned char *str, size_t len)
{
    __ASM volatile("SVC %0         \n"
                   "BX LR          \n"
                    : : "I" (2));
}
```

Impact

In IPC mode with PSA isolation level 2 and higher and `TFM_SP_LOG_RAW_ENABLED` option enabled an ARoT partition can expose to the stdout device any memory data using TF-M logging subsystem via SVC call.

Mitigation

Ensure that data sent for logging belongs to the current partition. For that purpose `tfm_hal_memory_check(curr_partition->boundary, data, size, TFM_HAL_ACCESS_READABLE)` is added to the logging function of the SVC handler. If the check fails then `tfm_core_panic()` is invoked and system halts.

ID	Title
TFMV-1	NS world may cause the CPU to perform an unexpected return operation due to unsealed stacks.
TFMV-2	Invoking Secure functions from handler mode may cause TF-M IPC model to behave unexpectedly.
TFMV-3	`abort()` function may not take effect in TF-M Crypto multi-part MAC/hashing/cipher operations.
TFMV-4	NSPE may access secure keys stored in TF-M Crypto service in Profile Small with Crypto key ID encoding disabled.
TFMV-5	`psa_fwu_write()` may cause buffer overflow in SPE.
TFMV-6	Partial tag comparison when using Chacha20-Poly1305 on the PSA driver API interface in CryptoCell enabled platforms
TFMV-7	ARoT can access PRoT data via debug logging functionality

---

3.3 Security Disclosures

Trusted Firmware-M(TF-M) disclose all security vulnerabilities, or are advised about, that are relevant to TF-M. TF-M encourage responsible disclosure of vulnerabilities and try the best to inform users about all possible issues.

The TF-M vulnerabilities are disclosed as Security Advisories, all of which are listed at the bottom of this page.

3.4 Found a Security Issue?

Although TF-M try to keep secure, it can only do so with the help of the community of developers and security researchers.

> **Warning:** If any security vulnerability was found, please **do not** report it in the issue tracker or on the mailing list. Instead, please follow the Security incident process.

One of the goals of this process is to ensure providers of products that use TF-M have a chance to consider the implications of the vulnerability and its remedy before it is made public. As such, please follow the disclosure plan outlined in the Security Incident Process. TF-M do the best to respond and fix any issues quickly.

Afterwards, write-up all the findings about the TF-M source code is highly encouraged.

3.5 Attribution

TF-M values researchers and community members who report vulnerabilities and TF-M policy is to credit the contributor's name in the published security advisory.

CHAPTER

FOUR

RELEASES

4.1 Version 2.0.0

4.1.1 New major features

- TF-M secure build process and non-secure build process are split to simplify and enhance non-secure integration with TF-M.
  - Refer to *TF-M Build Instruction* to build SPE image.
  - Refer to *Building Tests* to build non-secure tests.
- *Update new Mailbox agent API*.
  - Decouple the specific application Mailbox from SPM, make it an application in Secure Partition.
  - Unify the interfaces between partitions and SPM, and reduces the interaction interface between them.
- Multi-core support in the Secure Function (SFN) model.
- Optimize SPM critical section implementation to reduce time cost in isolation level 2&3.
- Use local variables for connection handles instead of dynamic allocation when there is only synchronous service access routine in the SFN model.
- P256-M[1] component is enabled on the TF-M side in profile medium which has a much smaller code size and RAM footprint.
- MCUboot upgrade to v2.0.0.
- Mbed TLS upgrade to v3.5.0.
- TF-M PSA client API performance profiling is tracked in SQUAD[2] and the profiling tool[3] is updated.
- TF-M integrates Read the Docs[4] to support finding documentation versions by the released tags and downloading PDFs. External links are supported for documentation in TF-M Tests, Tools and Extras repositories.

[1] P256-M

[2] TF-M QA Reports SQUAD

[3] TF-M performance profiling tool

[4] Read the Docs

4.1.2 New security advisories

A Security vulnerability fixed in v1.8.1 Refer to *TFMV-6* for more details. The mitigation is included in this release.

4.1.3 New platforms supported

- *stm32h573i_dk.*

4.1.4 Tested platforms

The following platforms are successfully tested in this release.

- **Arm**
  - AN519
  - AN521
  - AN555
  - Corstone-1000
  - Corstone-300
  - Corstone-310
  - Musca-B1
  - Musca-S1
- **Infineon/Cypress**
  - PSoC 64
- **STM**
  - B-U585I-IOT02A
  - NUCLEO-L552ZE-Q
  - STM32H573idk
- **Nordic**
  - nRF5340
  - nRF9160
- **NuMaker-PFM**
  - M2351
  - M2354
- **NXP**
  - LPCXpresso55S69

4.1.5 Reference memory footprint

All measurements below are made for *AN521* platform, built TF-Mv2.0.0-RC2 on Windows 10 using Armclang v6.18 and build type MinSizeRel.

All modules are measured in bytes. Some minor modules are not shown in the table below.

Note: Profile *Medium-ARoT-less* built with disabled Firmware Update service to align with other TF-M Profiles.

Module	Base		Small		ARoT-less		Medium		Large	
	Flash	RAM	Flash	RAM	Flash	RAM	Flash	RAM	Flash	RAM
Generated (stack, stc)	112	3184	160	3184	160	3184	208	3184	272	3184
Objects	940	1064	1224	5464	1313	6152	1443	1496	1518	1496
c_w.l	190	0	690	0	690	0	690	0	930	0
platform (Secure)	5098	284	5430	284	5782	284	6154	284	6284	284
SPM	3574	165	4456	165	3946	165	6330	1353	6484	1358
sprt	274	0	1470	0	1308	0	2470	4	2454	4
MbedCrypto	N/A	N/A	25220	2108	29964	2104	29968	2104	78938	1996
PROT_attestation	N/A	N/A	2341	557	2571	1218	2571	3010	2687	3010
PROT_crypto	N/A	N/A	3866	2070	4420	16026	4420	22938	4552	25818
PROT_ITS	N/A	N/A	4830	80	4894	112	5064	1988	5068	2498
PROT_platform	N/A	N/A	N/A	N/A	478	0	520	1280	520	1280
AROT_PS	N/A	N/A	N/A	N/A	N/A	N/A	3276	4364	3276	4364
platform_crypto_keys	N/A	N/A	248	0	256	0	256	0	256	0
qcbor	N/A	N/A	854	0	854	0	854	0	854	0
crypto_service_p256m	N/A	N/A	N/A	N/A	3534	0	3534	0	N/A	N/A
Padding	32	39	111	16	118	19	126	47	187	38
Total incl. padding	10220	4736	50900	13928	60288	29264	67884	42052	114280	45300

4.1.6 Known issues

Some open issues are not fixed in this release.

Descriptions	Issue links
TF-M Kconfig is broken due to build split. It will be recovered in a future release.	Not tracked

4.1.7 Issues fixed since v1.8.0

The following issues have been fixed since the v1.8.0 release.

Descriptions	Issue links
Arm GNU toolchain version greater than *11.2* has a linker issue in *syscall*.	https://developer.trustedfirmware.org/T1029

4.1.8 Reference

4.2 Version 1.8.1

4.2.1 Hotfixes

- Fix for "Partial tag comparison when using Chacha20-Poly1305 on the PSA driver API interface in CryptoCell enabled platforms" (CVE-2023-40271) (*TFMV-6*)

4.3 Version 1.8.0

4.3.1 New major features

- TF-M eRPC Test framework[1] is integrated.
- TF-M builtin key loader integration is reworked.
- Improved crypto library abstraction from TF-M Crypto service.
- *Kconfig system* is enhanced and finalized.
- Switch to upstream QCBOR[2].
- Enable PSA Authenticated Debug Access Control (ADAC)[3] support on Musca-B1 platform.
- Support Floating-Point (FP) with Arm Compiler.
- FF-M API uses signals to drive the partition scheduling instead of controlling partition context directly.
- MCUboot upgrade to v1.10.0.
- Mbed TLS upgrade to v3.4.0.
- Refine documentation restructure.
- It is optional to update copyright year in changes. Requirements of copyright note update is updated in *Contributing Process*.

[1] TF-M eRPC Test Framework

[2] QCBOR repository

[3] Authenticated Debug Access Control Specification

4.3.2 New platforms supported

- *AN555.*

4.3.3 Tested platforms

The following platforms are successfully tested in this release.

- AN519
- AN521
- AN555
- B-U585I-IOT02A
- Corstone-1000
- LPCXpresso55S69
- Musca-B1
- Musca-S1
- nRF9160
- nRF5340
- NUCLEO-L552ZE-Q
- PSoC 64
- RSS
- STM32L562E-DK

4.3.4 Reference memory footprint

All measurements below are made for *AN521* platform, built TF-Mv1.8.0-RC2 on Windows 10 using Armclang v6.18 and build type MinSizeRel.

All modules are measured in bytes. Some minor modules are not shown in the table below.

Note: Profile *Medium-ARoT-less* built with disabled Firmware Update service to algin with other TF-M Profiles.

Module	Base		Small		ARoT-less		Medium		Large	
	Flash	RAM	Flash	RAM	Flash	RAM	Flash	RAM	Flash	RAM
Generated (stack, stc)	112	3184	160	3184	160	3184	208	3184	272	3184
Objects	936	1064	1222	5464	1309	6152	1439	1544	1514	1544
c_w.l	190	0	568	0	808	0	808	0	808	0
platform (Secure)	5098	284	5530	284	5890	284	6256	284	6278	284
SPM	3694	1205	4560	577	4070	1205	5710	1261	5870	1266
sprt	254	0	1432	0	1270	0	2198	4	2198	4
MbedCrypto	N/A	N/A	24612	2292	46558	2296	46562	2296	80435	2408
PROT_attestation	N/A	N/A	2751	557	2981	1218	2981	3010	2981	3010
PROT_crypto	N/A	N/A	3686	2070	4240	16026	4240	22938	4368	24986
PROT_ITS	N/A	N/A	4826	80	4882	112	5056	1988	5068	2468
PROT_platform	N/A	N/A	N/A	N/A	480	0	522	1280	522	1280
AROT_PS	N/A	N/A	N/A	N/A	N/A	N/A	3340	6444	3340	6444
Padding	40	23	119	28	158	35	142	39	180	66
Total incl. padding	10324	5760	50576	14536	73916	30512	80572	44272	114924	46944

4.3.5 Known issues

Some open issues are not fixed in this release.

Descriptions	Issue links
Arm GNU toolchain version greater than *11.2* has a linker issue in *syscall*.	https://developer.trustedfirmware.org/T1029
In SFN model, input data size value in structure *psa_msg_t* passed to RoT Services might decrease after *psa_read()* or *psa_skip()*.	https://developer.trustedfirmware.org/T1030

4.3.6 Issues fixed since v1.7.0

The following issues have been fixed since v1.7.0 release.

Descriptions	Issue links
PSoC 64 platform builds and works well with ArmClang v6.13 and later.	Not tracked
LPCXpresso55S690 platform works well in profile small only.	Not tracked

4.3.7 Issues closed since v1.7.0

The following issues in v1.7.0 known issue list are closed. These issues are related to platform specific implementations or 3rd-party tools. Therefore they won't be fixed by TF-M.

Descriptions	Issue links
Latest CMake (3.25) has an issue with PSA Arch test	https://github.com/ARM-software/psa-arch-tests/issues/335

4.3.8 Reference

4.4 Version 1.7.0

4.4.1 New major features

- Unified API to PSA Service access implementing PSA FF-M[1] and FF-M v1.1 extension[2]
- Improve and simplify TF-M *configurability*
- Introduce the base configuration for TF-M essential
- Shift config options from CMake to C header files for PSA modules and platforms
- Employ Kconfig as a configuration tool and dependency tracker
- Configurable stack size for Secure Partitions
- Add *TF-M Medium-ARoT-less profile*[3]
- MM-IOVEC deployed in PSA Secure Partitions
- PSA FWU API updated to v1.0
- Mbed TLS updated to v3.2.1
- Code size further optimised. Please check *Reference footprint* section for details
- The manifest tool is decoupled with the build system
- Improvements in the Code size analysis tool
- Updated documentation

4.4.2 New platforms supported

- *Runtime Security Engine (RSE)*
- Platform *arm/mps3/Polaris* renamed to *Corstone-310*

4.4.3 Deprecated platforms and features

- TF-M Library model. Replaced by fully featured SFN model

[1] Firmware Framework for M (FF-M)
[2] Arm Firmware Framework for M 1.1 Extensions
[3] SESIP Profile for PSA Certified ARoT-less Level 2

4.4.4 Tested platforms

The following platforms are successfully tested in this release.

- AN519
- AN521
- AN547
- B-U585I-IOT02A
- Corstone-1000
- Corstone-310
- LPCXpresso55S69[6]
- Musca-B1
- Musca-S1
- nrf9160
- nrf5340
- NUCLEO-L552ZE-Q
- PSoC 64[6]
- RSS
- STM32L562E-DK

4.4.5 Reference footprint

All measurements below are made for *an521* platform, built TF-Mv1.7.0-RC3 on Windows 10 using Armclang v6.18 and build type MinSizeRel. Profile *Medium-ARoT-less* built with disabled Firmware Update service.

Module	Base		Small		ARoT-less		Medium		Large	
	Flash	RAM	Flash	RAM	Flash	RAM	Flash	RAM	Flash	RAM
Generated (stack, stc)	80	3184	128	3184	128	3184	176	3184	224	3184
Objects	896	1064	1180	5460	1269	6152	1391	1540	1463	1540
c_w.l	190	0	568	0	808	0	808	0	808	0
platform (Secure)	5046	284	5490	284	5842	284	6212	288	6234	288
SPM	3718	1333	4622	625	4094	1333	5672	1353	5832	1358
sprt	254	0	1432	0	1270	0	2004	4	2004	4
QCBOR			858	0	858	0	858	0	858	0
MbedCrypto			24606	2292	45140	2296	45140	2296	80827	2408
PROT_attestation			2751	557	2981	1218	2981	3010	2981	3010
PROT_crypto			3294	2692	3864	16648	3864	23560	3978	25608
PROT_ITS			4942	76	5042	108	5244	1984	5252	2464
PROT_platform					480	0	522	1280	522	1280
AROT_PS							3340	6444	3340	6444
platform_crypto_keys			348	0	356	0	356	0	356	0
Total incl. padding	10220	5892	50324	15216	72272	31264	78696	44984	114852	47608

[6] Please check the *Known issues* section for details.

4.4.6 Known issues

Some open issues are not fixed in this release.

Descriptions	Issue links
PSoC 64 platform builds and works well with ArmClang v6.13 and later.	Not tracked
LPCXpresso55S690 platform works well in profile small only.	Not tracked
Latest CMake (3.25) has an issue with PSA Arch test	https://github.com/ARM-software/psa-arch-tests/issues/335
PSA Arch framework test suite skip 2 test cases	PSA Arch framework test failures[4]

4.4.7 Issues closed since v1.6.0

Descriptions	Issue links
Psa Arch FF Test has no failures anymore but some tests are still skipped	PSA Arch FF Test Result[5]

4.4.8 Reference

4.5 Release Cadence and Process

4.5.1 Project Release Cadence

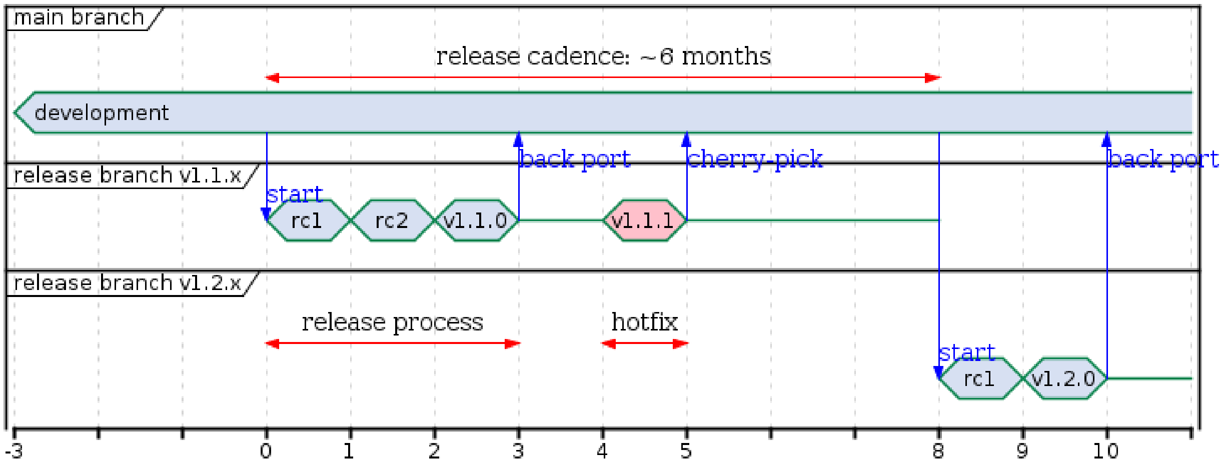
The project aims to do a release once every 6 months around April and November time and are listed in the *Future release plans*. The releases are to be performed and tagged on a dedicated release branch. The release process is initiated by an announcement in TF-M mailing list and followed by the creation of a release branch. Normal development on the main branch is not blocked and can be continued. The testing will be performed on release candidates and depending on issues found, additional candidates may be created to fix and retest the issues.

The Platform owners are expected to verify their platforms and confirm the correct operations or provide fixes in a timely manner to include in the release. The release notes will list all verified platforms. The platforms in Trusted Firmware OpenCI is automatically tested and any issues found shall be fixed.

After the final tag, the changes from the release branch will be back ported to main branch.

[4] PSA Arch framework test results

[5] PSA Arch FF Test Result



Although this document specifies the release cadence, this does not preclude an adhoc release for specific project requirements.

Note: When a new release starts the previous release branch obsoletes and is a subject for removal as shown for *v1.1.x* on the diagram above. At any moment only the latest release branch is maintained. The release tags will point to a commit in detached head state.

4.5.2 Release Version Scheme

Trusted Firmware-M uses a semantic versioning scheme. A version number is compiled as a dot separated set of numbers:

TF-Mv<MAJOR>.<MINOR>.<HOTFIX>

- <MAJOR>: Major release version for significant feature and API changes.
- <MINOR>: Minor release version for incremental features and API changes.
- <HOTFIX>: Used only for backporting **critical bug fix/security patches**.

Version	Date	PSA-arch tag/hash
v2.0.0	2023-11-28	v23.06_API1.5_ADAC_EAC
v1.8.1	2023-09-08	cf8bd71
v1.8.0	2023-04-28	cf8bd71
v1.7.0	2022-12-08	cf8bd71
v1.6.1	2022-11-16	f7e8495
v1.6.0	2022-04-28	f7e8495
v1.5.0	2021-11-30	v21.10_API1.3_ADAC_ALPHA-1
v1.4.0	2021-08-03	51ff2bd
v1.3.0	2021-04-07	b0635d9
v1.2.0	2020-11-25	90c8e680
v1.1	2020-07-15	1f960947
v1.0	2020-03-27	v20.03_API1.0

4.6 Future release plans

The dates below are tentative and subject to change.

Version	Feature Freeze	Release
v2.1.0	2024-04-5	2024-04-26

Please refer to *Release Cadence and Process* for interpreting version numbers.

CHAPTER

FIVE

ROADMAP

TF-M has been under active development since it was launched in Q1'18. It is being designed to include

1. Secure boot ensuring integrity of runtime images and responsible for firmware upgrade.
2. Runtime firmware consisting of TF-M Core is responsible for secure isolation, execution and communication aspects. and a set of Secure Services providing services to the Non-Secure and Secure Applications. The secures services currently supported are Secure Storage, Cryptography, Firmware Update, Attestation and Platform Services

If you are interested in collaborating on any of the roadmap features or other features, please mail TF-M mailing list

5.1 Supported Features

- PSA Firmware Framework v1.0, 1.1 Extension including IPC and SFN modes.
- PSA Level1, 2 and 3 Isolation.
- Secure Boot (mcuboot upstream) including generic fault injection mitigations
- PSA Protected Storage, Internal Trusted Storage v1.0 and Encrypted ITS
- PSA Cryptov1.0 (uses Mbed TLS v3.4.0)
- PSA Initial Attestation Service v1.0
- PSA Firmware Update v1.0
- PSA ADAC Specification Implementation
- Base Config, kconfig based configuration
- Profile Small, Medium, ARoT-less Medium, Large
- Secure Partition Interrupt Handling, Pre-emption of SPE execution
- Dual CPU
- Open Continuous Integration (CI) System
- Boot and Runtime Crypto Hardware Integration
- Fault Injection Handling library to mitigate against physical attacks
- Threat Model
- Arm v8.1-M Privileged Execute Never (PXN) attribute and Thread reentrancy disabled (TRD)
- FPU, MVE Support
- CC-312 PSA Cryptoprocessor Driver Interface

- Secure Storage - Key Diversification Enhancements
- Build System - Separate Secure and Non-Secure builds
- PSA Crypto layer for mcuboot/BL2

5.2 CQ2'24

- Supporting multiple clients (Hybrid Platforms) i.e. TF-M supporting multiple on core and off core clients on heterogeneous (e.g. Cortex-A + Cortex-M platforms) Mailbox API etc.
- Mbed TLS 3.6-LTS integration
- TF-M v2.1.0 Long Term Stable (LTS) Release
- Remote Test Infrastructure
- Arm v8.1-M Architecture Enablement - PAC/BTI

5.3 Future

- Implement support for multiple clients (Hybrid Platforms) contd.
- Build System Enhancements - Simplify build scripts
- MISRA testing/documentation
- TF-M Performance - Further Benchmarking and Optimization
- Scheduler - Multiple Secure Context Implementation
- PSA FWU Service Enhancements
- PSA ADAC Spec - Enhancements and Testing
- Arm v8.1-M Unprevileged Debug
- [Secure Storage] Extended PSA APIs
- [Audit Logs] Secure Storage, Policy Manager
- PSA FF Lifecycle API
- Fuzz Testing

CHAPTER

SIX

GLOSSARY OF TERMS AND ABBREVIATIONS

AAPCS
ARM Architecture Procedure Call Standard: The AAPCS defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine and a called routine

Application RoT
PSA term. The security domain in which additional security services are implemented. Also referred as ARoT.

HAL
Hardware Abstraction Layer: Interface to abstract hardware-oriented operations and provides a set of APIs to the upper layers.

ITS
Internal Trusted Storage
One of PSA services provided by TF-M.

MPC
Memory Protection Controller: Bus slave-side security controller for memory regions.

MPU
Memory Protection Unit: Hardware component providing privilege control.

NSPE
Non Secure Processing Enviroment: PSA term. In TF-M this means non secure domain typically running an OS using services provided by TF-M.

PPC
Peripheral Protection Controller: Bus slave-side security controller for peripheral access.

PS
Protected Storage
One of PSA services provided by TF-M.

PSA
PSA term. Platform Security Architecture.

PSA RoT
PSA term. This defines the most trusted security domain within a PSA system. Also referred as PRoT.

PSA-FF
PSA term. Platform Security Architecture Firmware Framework.

PSA-FF-M
PSA term. Platform Security Architecture Firmware Framework for M.

RoT
Root of Trust: PSA term. This is the minimal set of software, hardware and data that is implicitly trusted in the

platform — there is no software or hardware at a deeper level that can verify that the Root of Trust is authentic and unmodified.

RoT Service
PSA term. A set of related security operations that are implemented in a Secure Partition.

S/NS
Secure/Non-secure: The separation provided by TrustZone hardware components in the system.

SAU
Secure Attribution Unit: Hardware component providing isolation between Secure, Non-secure Callable and Non-secure addresses.

SFN
Secure Function: The function entry to a secure service. Multiple SFN per SS are permitted.

SP
Secure Partition
A logical container for secure services.

SPE
Secure Processing Environment: PSA term. In TF-M this means the secure domain protected by TF-M.

SPM
Secure Partition Manager
The TF-M component responsible for enumeration, management and isolation of multiple Secure Partitions within the TEE.

SPRT
Secure Partition Runtime: The TF-M component responsible for Secure Partition runtime functionalities.

SPRTL
Secure Partition Runtime Library: A library contains the SPRT code and data.

SS
Secure Service: A component within the TEE that is atomic from a security/trust point of view, i.e. which is viewed as a single entity from a TF-M point of view.

SVC
SuperVisor Call: ARMv7M assembly instruction to call a privileged handler function

TBSA-M
Trusted Base System Architecture for M. TBSA term. See Trusted Base System Architecture for M

TFM
TF-M
Trusted Firmware-M or Trusted Firmware for M-class. ARM TF-M provides a reference implementation of secure world software for ARMv8-M.

Reference

Firmware Framework for M (FF-M)

Trusted Base System Architecture for M

CHAPTER

SEVEN

TF-M PLATFORMS

7.1 Arm platforms

7.1.1 Corstone-1000

Introduction

The ARM's Corstone-1000 platform is a reference implementation of PSA FF-M architecture where NSPE and SPE environments are partitioned/isolated into Cortex-A35 and Cortex-M0+ respectively.

Cortex-M0+ acting as Secure Enclave is the Root-of-trust of SoC. Its software comprises of two boot loading stages, i.e. Bl1 and Bl2 (based on mcuboot) and TF-M as run time software. Cortex-A35, also referred as host, is treated as non-secure from the Secure Enclave perspective. The Cortex-A35 is brought out of rest by Secure Enclave in aarch64 bit mode, and boots the software ecosystem based on linux, u-boot, UEFI run time services, TF-A, Secure Partitions and Optee.

The communication between NSPE and SPE is based on PSA IPC protocol running on top of the RSE communication protocol. The Corstone-1000 supports only the *Embed protocol*, and the ATU support is removed.

The secure enclave subsystem has ARM's CC-312 (Crypto Cell) hardware to accelerate cryptographic operations. Additionaly, platform supports Secure Debug using SDC-600 as the communication interface between host debugger and platform target. The platform has the build option to enable secure debug protocol to unlock debug ports during boot time. The protocol is based on ARM's ADAC (Authenticated Debug Access Control) standard.

System boot

- The SoC reset brings Secure Enclave (SE), that is Cortex-M0+, out of rest.
- SE executes the BL1 ROM code based on mcuboot.
- BL1 load, verifies and transfer execution to BL2 which is again based on mcuboot.
- BL2 loads and verifies TF-M and host's initial boot loader image.
- BL2 transfer the execution to the TF-M.
- During TF-M initialization, the host is taken out of rest.
- Hashes of the keys used for image verification are stored in the OTP memory.

Build

Platform solution

The platform binaries are build using Yocto. Below is the user guide:

Arm Corstone-1000 User Guide

Secure Test

This section can be used to test the secure enclave software independently from the host. The below configuration builds the secure enclave binaries with CI test frame integrated. On boot, secure enclave softwares stack is brought up, and CI tests starts executing at the end of the initialization process. In the below configuration, host software support is disabled, and meant only to test/verify the secure enclave softwares.

FVP

- Download Corstone-1000 FVP from : Arm Ecosystem FVPs
- Install FVP by running the shell script.
- Running of the binary will boot secure enclave software stack and at the end all CI test from tf-m-test along with platform specific tests are executed.

```
cd <tf-m-root>/
cmake -B build/ -S . -DCMAKE_BUILD_TYPE=Debug -DTFM_TOOLCHAIN_FILE=<tf-m-root>/toolchain_
↪GNUARM.cmake -DTFM_PLATFORM=arm/corstone1000 -DPLATFORM_IS_FVP=TRUE -DTEST_NS=OFF -
↪DTEST_S=ON -DEXTRA_S_TEST_SUITE_PATH=platform/ext/target/arm/corstone1000/ci_
↪regression_tests/
cmake --build build -- install
dd conv=notrunc bs=1 if=build/install/outputs/bl1_1.bin of=build/install/outputs/bl1.bin␣
↪seek=0
dd conv=notrunc bs=1 if=build/install/outputs/bl1_provisioning_bundle.bin of=build/
↪install/outputs/bl1.bin seek=40960
./platform/ext/target/arm/corstone1000/create-flash-image.sh build/install/outputs␣
↪cs1000.bin

<path-to-FVP-installation>/models/Linux64_GCC-9.3/FVP_Corstone-1000 -C board.
↪flashloader0.fname="none" -C se.trustedBootROMloader.fname="build/install/outputs/bl1.
↪bin" -C board.xnvm_size=64 -C se.trustedSRAM_config=6 -C se.BootROM_config="3" -C␣
↪board.smsc_91c111.enabled=0  -C board.hostbridge.userNetworking=true --data board.
↪flash0=build/install/outputs/cs1000.bin@0x68000000 -C diagnostics=4 -C disable_
↪visualisation=true -C board.se_flash_size=8192 -C diagnostics=4  -C disable_
↪visualisation=true
```

FPGA

- Follow the above pointed platform user guide to setup the FPGA board.
- Use the BL1 generated from the below commands to place it inside FPGA board SD Card.
- Use the cs1000.bin created from the below commands to place it inside FPGA board SD Card.

```
cd <tf-m-root>/
cmake -B build/ -S . -DCMAKE_BUILD_TYPE=Debug -DTFM_TOOLCHAIN_FILE=<tf-m-root>/toolchain_
↪GNUARM.cmake -DTFM_PLATFORM=arm/corstone1000 -DTEST_NS=OFF -DTEST_S=ON -DTEST_S_PS=OFF␣
↪-DTEST_S_PLATFORM=OFF -DEXTRA_S_TEST_SUITE_PATH=platform/ext/target/arm/corstone1000/
↪ci_regression_tests/
cmake --build build -- install
dd conv=notrunc bs=1 if=build/install/outputs/bl1_1.bin of=build/install/outputs/bl1.bin␣
↪seek=0
dd conv=notrunc bs=1 if=build/install/outputs/bl1_provisioning_bundle.bin of=build/
↪install/outputs/bl1.bin seek=40960
./platform/ext/target/arm/corstone1000/create-flash-image.sh build/install/outputs␣
↪cs1000.bin
cp build/install/outputs/bl1.bin <path-to-FPGA-SD-CARD>/SOFTWARE/
cp build/install/outputs/cs1000.bin <path-to-FPGA-SD-CARD>/SOFTWARE/
```

FPGA build can not compile all the CI tests into a single build as it exceeds the available RAM size. So there is a need to select few tests but not all. The above configuration disable build of -DTEST_S_PS and -DTEST_S_PLATFORM. Other test configurations are:

- -DTEST_S_ATTESTATION=ON/OFF
- -DTEST_S_CRYPTO=ON/OFF
- -DTEST_S_ITS=ON/OFF
- -DTEST_S_PS=ON/OFF
- -DTEST_S_PLATFORM=ON/OFF

7.1.2 Corstone SSE-300 with Ethos-U55/U65 Example Subsystem for Arm Ecosystem FVP and for MPS3 (AN547, AN552)

Introduction

Corstone-300 is an Arm reference subsystem for secure System on Chips containing an Armv8.1-M Cortex-M55 processor and an Ethos-U55/U65 neural network processor. It is an MPS3 based platform with the usual MPS3 peripherals.

This platform port supports all TF-M regression tests (Secure and Non-secure) with Isolation Level 1 and 2.

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Note: For Armclang compiler v6.18 or later version is required.

Building TF-M

Follow the instructions in Building instructions.

Build instructions with platform name: arm/mps3/corstone300/an547

```
-DTFM_PLATFORM=arm/mps3/corstone300/an547
```

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Build instructions with platform name: arm/mps3/corstone300/an552

```
-DTFM_PLATFORM=arm/mps3/corstone300/an552
```

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Build instructions with platform name: arm/mps3/corstone300/fvp

```
-DTFM_PLATFORM=arm/mps3/corstone300/fvp
```

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Note: The built binaries can be run on the Corstone-300 Ethos-U55/U65 Ecosystem FVP (FVP_SSE300_MPS3). At least Ecosystem FVP version 11.22 is required.

Note: Provisioning bundles can be generated with the `-DPLATFORM_DEFAULT_PROVISIONING=OFF` flag. The provisioning bundle binary will be generated and it's going to contain the provisioning code and provisioning values.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` and `-DTFM_DUMMY_PROVISIONING=ON` then the keys in the `tf-m/platform/ext/target/arm/mps3/common/provisioning/provisioning_config.cmake` and the default MCUBoot signing keys will be used for provisioning.

If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` and `-DTFM_DUMMY_PROVISIONING=OFF` are set then unique values can be used for provisioning. The keys and seeds can be changed by passing the new values to the build command, or by setting the `-DPROVISIONING_KEYS_CONFIG` flag to a .cmake file that contains the keys. An example config cmake file can be seen at `tf-m/platform/ext/target/arm/mps3/common/provisioning/provisioning_config.cmake`. Otherwise new random values are going to be generated and used. For the image signing the ${MCUBOOT_KEY_S} and ${MCUBOOT_KEY_NS} will be used. These variables should point to .pem files that contain the code signing private keys. The public keys are going to be generated from these private keys and will be used for provisioning. The hash of the public key is going to be written into the `provisioning_data.c` automatically.

If -DMCUBOOT_GENERATE_SIGNING_KEYPAIR=ON is set then a new mcuboot signing public and private keypair is going to be generated and it's going to be used to sign the S and NS binaries.

The new generated keypair can be found in the <build dir>/bin folder or in the <install directory>/image_signing/keys after installation. The generated provisioning_data.c file can be found at <build directory>/platform/target/provisioning/provisioning_data.c

Note: The provisioning bundle generation depends on pyelftools that's have to be installed:

```
pip3 install pyelftools
```

To run the example code on Corstone SSE-300 with Ethos-U55/U65 Example Subsystem for MPS3 (AN547)

FPGA image is available to download here

If the link above is not working just go to Arm PDH and search for AN547.

To run BL2 bootloader, TF-M example application and tests in the MPS3 board, it is required to have AN547 image in the MPS3 board SD card. The image should be located in <MPS3 device name>/MB/HBI<BoardNumberBoardrevision>/AN547

The MPS3 board tested is HBI0309C.

1. Copy bl2.bin and tfm_s_ns_signed.bin files from build dir to <MPS3 device name>/SOFTWARE/
2. Rename tfm_s_ns_signed.bin to tfm.bin (Filename should not be longer than 8 charachters.)
3. Open <MPS3 device name>/MB/HBI0309C/AN547/images.txt
4. Update the images.txt file as follows:

```
TITLE: Arm MPS3 FPGA prototyping board Images Configuration File

[IMAGES]
TOTALIMAGES: 2                      ;Number of Images (Max: 32)

IMAGE0UPDATE: AUTO                  ;Image Update:NONE/AUTO/FORCE
IMAGE0ADDRESS: 0x00000000           ;Please select the required executable program
IMAGE0FILE: \SOFTWARE\bl2.bin
IMAGE1UPDATE: FORCEQSPI
IMAGE1ADDRESS: 0x00000000
IMAGE1FILE: \SOFTWARE\tfm.bin
```

5. Close <MPS3 device name>/MB/HBI0309C/AN547/images.txt
6. Unmount/eject the <MPS3 device name> unit
7. Reset the board to execute the TF-M example application
8. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
```

(continues on next page)

(continued from previous page)

```
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[WRN] This device was provisioned with dummy keys. This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.8.1
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x10022400` address by copying `encrypted_provisioning_bundle.bin` and renaming it to `prv.bin`, then extending the images.txt with:

```
IMAGE2UPDATE: AUTO
IMAGE2ADDRESS: 0x00022400
IMAGE2FILE: \SOFTWARE\prv.bin
```

To run the example code on Corstone SSE-300 with Ethos-U55/U65 Example Subsystem for MPS3 (AN552)

FPGA image is available to download here

If the link above is not working just go to Arm PDH and search for AN552.

To run BL2 bootloader and TF-M example application and tests in the MPS3 board, it is required to have AN552 image in the MPS3 board SD card. The image should be located in `<MPS3 device name>/MB/HBI<BoardNumberBoardrevision>/AN552`

The MPS3 board tested is HBI0309C.

1. Copy `bl2.bin` and `tfm_s_ns_signed.bin` files from build dir to `<MPS3 device name>/SOFTWARE/`
2. Rename `tfm_s_ns_signed.bin` to `tfm.bin` (Filename should not be longer than 8 charachters.)
3. Open `<MPS3 device name>/MB/HBI0309C/AN552/images.txt`
4. Update the `images.txt` file as follows:

```
TITLE: Arm MPS3 FPGA prototyping board Images Configuration File

[IMAGES]
TOTALIMAGES: 2                    ;Number of Images (Max: 32)

IMAGE0UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
```

(continues on next page)

(continued from previous page)

```
IMAGE0ADDRESS: 0x00000000          ;Please select the required executable program
IMAGE0FILE: \SOFTWARE\bl2.bin
IMAGE1UPDATE: FORCEQSPI
IMAGE1ADDRESS: 0x00000000
IMAGE1FILE: \SOFTWARE\tfm.bin
```

5. Close `<MPS3 device name>/MB/HBI0309C/AN552/images.txt`
6. Unmount/eject the `<MPS3 device name>` unit
7. Reset the board to execute the TF-M example application
8. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[WRN] This device was provisioned with dummy keys. This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.8.1
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x10022400` address by copying `encrypted_provisioning_bundle.bin` and renaming it to `prv.bin`, then extending the images.txt with:

```
IMAGE2UPDATE: AUTO
IMAGE2ADDRESS: 0x00022400
IMAGE2FILE: \SOFTWARE\prv.bin
```

To run the example code on Corstone-300 Ethos-U55/U65 Ecosystem FVP

FVP is available to download here

1. Install the FVP
2. Copy `bl2.axf` and `tfm_s_ns_signed.bin` files from build dir to `<FVP installation path>/models/Linux64_GCC-9.3/`
3. Navigate to the same directory and execute the following command to start FVP:

```
$ ./FVP_Corstone_SSE-300_Ethos-U55 -a cpu0*="bl2.axf" --data "tfm_s_ns_signed.bin
↪"@0x38000000
```

4. After completing the procedure you should be able to see similar messages to this on the serial port (baud 115200 8n1):

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[WRN] This device was provisioned with dummy keys. This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.8.1
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x10022000` address with:

```
$ ./FVP_Corstone_SSE-300_Ethos-U55 -a cpu0*="<path-to-build-directory>/bl2.axf" --data "
↪<path-to-build-directory>/tfm_s_ns_signed.bin"@0x38000000 --data "<path-to-build-
↪directory>/encrypted_provisioning_bundle.bin"@0x10022000
```

7.1.3 Corstone SSE-310 with Ethos-U55/U65 Example Subsystem for Arm Ecosystem FVP and for MPS3 (AN555)

Introduction

Corstone-310 (formerly Corstone-Polaris) is an Arm reference subsystem for secure System on Chips containing an Armv8.1-M Cortex-M85 processor and an Ethos-U55/U65 neural network processor. It is an MPS3 based platform with the usual MPS3 peripherals.

This platform port supports all TF-M regression tests (Secure and Non-secure) with Isolation Level 1 and 2.

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Note: For Armclang compiler v6.18 or later version is required.

Building TF-M

Follow the instructions in Building instructions.

Build instructions with platform name: arm/mps3/corstone310/an555

```
-DTFM_PLATFORM=arm/mps3/corstone310/an555
```

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Build instructions with platform name: arm/mps3/corstone310/fvp

```
-DTFM_PLATFORM=arm/mps3/corstone310/fvp
```

Note: For Ethos-U55/U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Note: The built binaries can be run on the Corstone-310 Ethos-U55/U65 Ecosystem FVP (FVP_SSE310_MPS3). At least Ecosystem FVP version 11.22 is required.

Note: Provisioning bundles can be generated with the `-DPLATFORM_DEFAULT_PROVISIONING=OFF` flag. The provisioning bundle binary will be generated and it's going to contain the provisioning code and provisioning values.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` and `-DTFM_DUMMY_PROVISIONING=ON` then the keys in the tf-m/platform/ext/target/arm/mps3/common/provisioning/provisioning_config.cmake and the default MCUBoot signing keys will be used for provisioning.

If -DPLATFORM_DEFAULT_PROVISIONING=OFF and -DTFM_DUMMY_PROVISIONING=OFF are set then unique values can be used for provisioning. The keys and seeds can be changed by passing the new values to the build command, or by setting the -DPROVISIONING_KEYS_CONFIG flag to a .cmake file that contains the keys. An example config cmake file can be seen at tf-m/platform/ext/target/arm/mps3/common/provisioning/provisioning_config.cmake. Otherwise new random values are going to be generated and used. For the image signing the ${MCUBOOT_KEY_S} and ${MCUBOOT_KEY_NS} will be used. These variables should point to .pem files that contain the code signing private keys. The public keys are going to be generated from these private keys and will be used for provisioning. The hash of the public key is going to be written into the provisioning_data.c automatically.

If -DMCUBOOT_GENERATE_SIGNING_KEYPAIR=ON is set then a new mcuboot signing public and private keypair is going to be generated and it's going to be used to sign the S and NS binaries.

The new generated keypair can be found in the <build dir>/bin folder or in the <install directory>/image_signing/keys after installation. The generated provisioning_data.c file can be found at <build directory>/platform/target/provisioning/provisioning_data.c

Note: The provisioning bundle generation depends on pyelftools that's have to be installed:

```
pip3 install pyelftools
```

To run the example code on Corstone SSE-310 with Ethos-U55/U65 Example Subsystem for MPS3 (AN555)

FPGA image is available to download here

If the link above is not working just go to Arm PDH and search for AN555.

To run BL2 bootloader, TF-M example application and tests in the MPS3 board, it is required to have AN555 image in the MPS3 board SD card. The image should be located in <MPS3 device name>/MB/HBI<BoardNumberBoardrevision>/AN555

The MPS3 board tested is HBI0309C.

1. Execute the following command to create the tfm.bin binary which fills the entire available space on the MPS3 onboard QSPI. This way the whole flash content (PS, ITS, OTP, NV counters) is in a known state. (If anything left in the QSPI, this binary will overwrite it.) Also, MPS3 can only handle SFN 8.3 format, so the binary name must be shortened.:

   ```
   $ cd <build dir>/bin
   $ cp tfm_s_ns_signed.bin tfm.bin
   $ truncate -s 8M tfm.bin
   ```

2. Copy bl2.bin and tfm.bin files from build dir to <MPS3 device name>/SOFTWARE/
3. Open <MPS3 device name>/MB/HBI0309C/AN555/images.txt
4. Update the images.txt file as follows:

   ```
   [IMAGES]
   TOTALIMAGES: 2

   IMAGE0ADDRESS: 0x01_00_1100_0000
   ```

(continues on next page)

(continued from previous page)

```
IMAGE0UPDATE: RAM
IMAGE0FILE: \SOFTWARE\bl2.bin

IMAGE1ADDRESS: 0x01_00_0000_0000
IMAGE1UPDATE: FORCEQSPI
IMAGE1FILE: \SOFTWARE\tfm.bin
```

5. Close `<MPS3 device name>/MB/HBI0309C/AN555/images.txt`
6. Unmount/eject the `<MPS3 device name>` unit
7. Reset the board to execute the TF-M example application
8. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[WRN] This device was provisioned with dummy keys. This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.8.1
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x11022400` address by copying `encrypted_provisioning_bundle.bin` and renaming it to `prv.bin`, then extending the images.txt with:

```
IMAGE2UPDATE: RAM
IMAGE2ADDRESS: 0x01_00_1102_2400
IMAGE2FILE: \SOFTWARE\prv.bin
```

To run the example code on Corstone-310 Ethos-U55/U65 Ecosystem FVP

FVP is available to download here

1. Install the FVP
2. Copy `bl2.axf` and `tfm_s_ns_signed.bin` files from build dir to `<FVP installation path>/models/Linux64_GCC-9.3/`
3. Navigate to the same directory and execute the following command to start FVP:

```
$ ./FVP_Corstone_SSE-310 -a cpu0*="bl2.axf" --data "tfm_s_ns_signed.bin"@0x38000000
```

4. After completing the procedure you should be able to see similar messages to this on the serial port (baud 115200 8n1):

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[WRN] This device was provisioned with dummy keys. This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.8.1
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x11022000` address with:

```
$ ./FVP_Corstone_SSE-310 -a cpu0*="<path-to-build-directory>/bl2.axf" --data "<path-to-
↪build-directory>/tfm_s_ns_signed.bin"@0x38000000 --data "<path-to-build-directory>/
↪encrypted_provisioning_bundle.bin"@0x11022000
```

7.1.4 Corstone SSE-315 with Ethos-U65 Example Subsystem for Ecosystem FVP

Introduction

Corstone-315 is an Arm reference subsystem for secure System on Chips containing an Armv8.1-M Cortex-M85 processor and LCM, KMU and SAM IPs. and an Ethos-U65 neural network processor. It is an MPS4 based platform.

This platform port supports all TF-M regression tests (Secure and Non-secure) with Isolation Level 1 and 2.

Note: For Ethos-U65 IP this platform support only provides base address, interrupt number and an example NPU setup as non-secure, unprivileged.

Note: For Armclang compiler v6.18 or later version is required.

Building TF-M

Follow the instructions in Building instructions.

Build instructions with platform name: arm/mps4/corstone315

```
-DTFM_PLATFORM=arm/mps4/corstone315
```

Note: The built binaries can be run on the Corstone-315 Ethos-U65 Ecosystem FVP (FVP_Corstone_SSE-315).

Note: If `-DTFM_DUMMY_PROVISIONING=ON` then the keys in the `tf-M/platform/ext/target/arm/mps4/corstone315/provisioning/bundle_cm/cm_provisioning_config.cmake`, `tf-M/platform/ext/target/arm/mps4/corstone315/provisioning/bundle_dm/dm_provisioning_config.cmake` and default MCUBoot signing and encryption keys will be used for provisioning.

If `-DTFM_DUMMY_PROVISIONING=OFF` is set then unique values can be used for provisioning. The keys and seeds can be changed by passing the new values to the build command, or by setting the `-DDM_PROVISIONING_KEYS_CONFIG` and `-DCM_PROVISIONING_KEYS_CONFIG`` flag to the .cmake files which contain the keys. An example config cmake file can be seen at ``tf-m/platform/ext/target/arm/mps4/corstone315/provisioning/bundle_cm/cm_provisioning_config.cmake` and `tf-m/platform/ext/target/arm/mps4/corstone315/provisioning/bundle_dm/dm_provisioning_config.cmake`. Otherwise new random values are going to be generated and used (or default values in some cases). For the image signing the ${MCUBOOT_KEY_S} and ${MCUBOOT_KEY_NS} will be used. These variables should point to .pem files that contain the code signing private keys. The public keys are going to be generated from these private keys and will be used for provisioning. The hash of the public key is going to be written into the `provisioning_data.c` automatically. For the BL2 image encryption the `TFM_BL2_ENCRYPTION_KEY_PATH` has to be set to a .bin file which contains the encryption key. (For an example, see `tf-m/bl1/bl1_2/bl2_dummy_encryption_key.bin`)

If `-DMCUBOOT_GENERATE_SIGNING_KEYPAIR=ON` is set then a new MCUBoot signing public and private keypair is going to be generated and it's going to be used to sign the S and NS binaries.

The new generated keypair can be found in the `<build dir>/bin` folder or in the `<install directory>/image_signing/keys` after installation. The generated provisioning_data.c file can be found at `<build directory>/platform/target/provisioning/provisioning_data.c`

Note: The provisioning bundle generation depends on pyelftools that's have to be installed:

```
pip3 install pyelftools
```

To run the example code on Corstone-315 Ethos-U65 Ecosystem FVP

FVP is available to download here

1. Install the FVP
2. Copy `bl1_1.bin`, `cm_provisioning_bundle.bin`, `dm_provisioning_bundle.bin`, `bl2_signed.bin` and `tfm_s_ns_signed.bin` files from build dir to `<FVP installation path>/models/Linux64_GCC-6.4/`
3. Navigate to the same directory and execute the following command to start FVP:

```
$ ./FVP_Corstone_SSE-315 --data "bl1_1.bin"@0x11000000
               --data "cm_provisioning_bundle.bin"@0x12024000
               --data "dm_provisioning_bundle.bin"@0x1202aa00
               --data "bl2_signed.bin"@0x12031400
               --data "tfm_s_ns_signed.bin"@0x38000000
```

4. After completing the procedure you should be able to see similar messages to this on the serial port (baud 115200 8n1):

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[INF] Starting TF-M BL1_1
[INF] Beginning provisioning
[INF] TP mode set complete, system will now reset.
[INF] Starting TF-M BL1_1
[INF] Beginning provisioning
[INF] Waiting for CM provisioning bundle
[INF] Enabling secure provisioning mode, system will now reset.
[INF] Starting TF-M BL1_1
[INF] Beginning provisioning
[INF] Waiting for CM provisioning bundle
[INF] Running CM provisioning bundle
[INF] Starting TF-M BL1_1
[INF] Beginning provisioning
[INF] Waiting for DM provisioning bundle
[INF] Enabling secure provisioning mode, system will now reset.
[INF] Starting TF-M BL1_1
[INF] Beginning provisioning
[INF] Waiting for DM provisioning bundle
[INF] Running DM provisioning bundle
[INF] Starting TF-M BL1_1
[INF] Jumping to BL1_2
[INF] starting TF-M bl1_2
[INF] Attempting to boot image 0
[INF] BL2 image decrypted successfully
```

(continues on next page)

(continued from previous page)

```
[INF] BL2 image validated successfully
[INF] Jumping to BL2
[INF] Starting bootloader
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
Booting TF-M v2.0.0+d09934721
Non-Secure system starting...
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

7.1.5 Musca-B1 Platform Specifics

DAPLink Firmware

The code on Musca-B1 is running from embedded flash. Make sure that the DAPLink FW for eFlash is downloaded to the board. You can find on the Arm Community page A short description of how to update the DAPLink FW can be found there as well.

Note: Warm reset of eFlash is not supported on Musca_B1. TF-M may not boot after a warm reset. Further information on the hardware limitation can be found on Arm Community page.

Platform pin service

This service is designed to perform secure pin services of the platform (e.g alternate function setting, pin mode setting, etc). The service uses the IOCTL API of TF-M's Platform Service, which allows the non-secure application to make pin service requests on Musca B1 based on a generic service request delivery mechanism.

7.1.6 Musca-S1 Platform Specifics

DAPLink Firmware

The code on Musca-S1 is running from on-chip eMRAM (2MB non-volatile bootable memory). Make sure that the DAPLink FW for QSPI/MRAM is downloaded to the board. You can find on the Arm Community page A short description of how to update the DAPLink FW can be found there as well.

Platform pin service

This service is designed to perform secure pin services of the platform (e.g alternate function setting, pin mode setting, etc). The service uses the IOCTL API of TF-M's Platform Service, which allows the non-secure application to make pin service requests on Musca S1 based on a generic service request delivery mechanism.

7.1.7 Runtime Security Engine (RSE)

Previously known as Runtime Security Subsystem (RSS).

Runtime Security Engine (RSE)

Introduction

Runtime Security Engine (RSE) is an Arm subsystem that provides a reference implementation of the HES Host in the Arm Confidential Compute Architecture (CCA). It is designed to be integrated into A-profile compute subsystems that implement Arm CCA, where it serves as the Root of Trust.

RSE initially boots from immutable code (BL1_1) in its internal ROM, before jumping to BL1_2, which is provisioned and hash-locked in RSE OTP. The updatable MCUBoot BL2 boot stage is loaded from host system flash into RSE SRAM, where it is authenticated. BL2 loads and authenticates the TF-M runtime into RSE SRAM from host flash. BL2 is also responsible for loading initial boot code into other subsystems within the host.

The RSE platform port supports the TF-M Crypto, TF-M Initial Attestation, Measured Boot and TF-M Platform services along with the corresponding regression tests. It supports the IPC model in multi-core topology with Isolation Level 1 and 2.

Building TF-M

Follow the instructions in *Build instructions*. Build TF-M with platform name: *arm/rse/<rse platform name>*

For example for building RSE for Total Compute platforms: `-DTFM_PLATFORM=arm/rse/tc`

Signing host images

RSE BL2 can load boot images into other subsystems within the host system. It expects images to be signed, with the signatures attached to the images in the MCUBoot metadata format.

The imgtool Python package can be used to sign images in the required format. To sign a host image using the development key distributed with TF-M, use the following command:

```
imgtool sign \
    -k <TF-M base directory>/bl2/ext/mcuboot/root-RSA-3072.pem \
    --public-key-format full \
    --max-align 8 \
    --align 1 \
    -v "0.0.1" \
    -s 1 \
```

(continues on next page)

(continued from previous page)

```
    -H 0x2000 \
    --pad-header \
    -S 0x80000 \
    --pad \
    -L <load address> \
    <binary infile> \
    <signed binary outfile>
```

The `load address` is the logical address in the RSE memory map to which BL2 will load the image. RSE FW expects the first host image to be loaded to address `0x70000000` (the beginning of the RSE ATU host access region), and each subsequent host image to be loaded at an offset of `0x1000000` from the previous image. The RSE ATU should be configured to map these logical addresses to the physical addresses in the host system that the images need to be loaded to.

For more information on the `imgtool` parameters, see the MCUBoot imgtool documentation.

> **Warning:** The TF-M development key must never be used in production. To generate a production key, follow the imgtool documentation.

Running the code

To run the built images, first the ROM image must be created from the bl1_1 binary and the ROM DMA Initial Command Sequence (ICS).:

```
srec_cat \
        bl1_1.bin -Binary     -offset 0x0 \
        rom_dma_ics.bin -Binary -offset 0x1F000 \
        -o rom.bin -Binary
```

Then, the flash image must be created by concatenating the images that are output from the build. To create the flash image, the following `fiptool` command should be run. `fiptool` documentation can be found here. Note that an up-to-date fiptool that supports the RSE UUIDs must be used.:

```
fiptool create \
    --align 8192 --rss-bl2     bl2_signed.bin \
    --align 8192 --rss-ns      tfm_ns_signed.bin \
    --align 8192 --rss-s       tfm_s_signed.bin \
    --align 8192 --rss-scp-bl1 <signed Host SCP BL1 image> \
    --align 8192 --rss-ap-bl1  <signed Host AP BL1 image> \
    fip.bin
```

If you already have a `fip.bin` containing host firmware images, RSE FIP images can be patched in:

```
fiptool update --align 8192 --rss-bl2 bl2_signed.bin fip.bin
fiptool update --align 8192 --rss-ns  tfm_ns.bin fip.bin
fiptool update --align 8192 --rss-s   tfm_s.bin fip.bin
```

If XIP mode is enabled, the following `fiptool` command should be run to create the flash image:

```
fiptool create \
    --align 8192 --rss-bl2           bl2_signed.bin \
```

(continues on next page)

(continued from previous page)

```
    --align 8192 --rss-ns            tfm_ns_encrypted.bin \
    --align 8192 --rss-s             tfm_s_encrypted.bin \
    --align 8192 --rss-sic-tables-ns tfm_ns_sic_tables_signed.bin \
    --align 8192 --rss-sic-tables-s  tfm_s_sic_tables_signed.bin \
    --align 8192 --rss-scp-bl1       <signed Host SCP BL1 image> \
    --align 8192 --rss-ap-bl1        <signed Host AP BL1 image> \
    fip.bin
```

Once the FIP is prepared, a host flash image can be created using `srec_cat`:

```
srec_cat \
        fip.bin -Binary -offset 0x0 \
        -o host_flash.bin -Binary
```

If GPT support is enabled, and a host `fip.bin` and `fip_gpt.bin` has been obtained, RSE images can be inserted by first patching the host FIP and then inserting that patched FIP into the GPT image:

```
sector_size=$(gdisk -l fip_gpt.bin | grep -i "sector size (logical):" | \
            sed 's/.*logical): \([0-9]*\) bytes/\1/')

fip_label=" FIP_A$"
fip_start_sector=$(gdisk -l fip_gpt.bin | grep "$fip_label" | awk '{print $2}')
fip_sector_am=$(gdisk -l fip_gpt.bin | grep "$fip_label" | awk '{print $3 - $2}')

dd if=fip.bin of=fip_gpt.bin bs=$sector_size seek=$fip_start_sector \
    count=$fip_sector_am conv=notrunc

fip_label = " FIP_B$"
fip_start_sector = $(gdisk -l fip_gpt.bin | grep "$fip_label" | awk '{print $2}')
fip_sector_am = $(gdisk -l fip_gpt.bin | grep "$fip_label" | awk '{print $3 - $2}')

dd if=fip.bin of=fip_gpt.bin bs=$sector_size seek=$fip_start_sector \
    count=$fip_sector_am conv=notrunc
```

To patch a `fip_gpt.bin` without having an initial `fip.bin`, the FIP can be extracted from the GPT image using the following commands (and can then be patched and reinserted using the above commands):

```
sector_size=$(gdisk -l fip_gpt.bin | grep -i "sector size (logical):" | \
            sed 's/.*logical): \([0-9]*\) bytes/\1/')

fip_label=" FIP_A$"
fip_start_sector=$(gdisk -l fip_gpt.bin | grep "$fip_label" | awk '{print $2}')
fip_sector_am=$(gdisk -l fip_gpt.bin | grep "$fip_label" | awk '{print $3 - $2}')

dd if=fip_gpt.bin of=fip.bin bs=$sector_size skip=$fip_start_sector \
    count=$fip_sector_am conv=notrunc
```

Once the `fip_gpt.bin` is prepared, it is placed at the base of the host flash image:

```
srec_cat \
        fip_gpt.bin -Binary -offset 0x0 \
        -o host_flash.bin -Binary
```

The RSE ROM binary should be placed in RSE ROM at `0x11000000` and the host flash binary should be placed at the base of the host flash. For the TC platform, this is at `0x80000000`.

The RSE OTP must be provisioned. On a development platform with `TFM_DUMMY_PROVISIONING` enabled, BL1_1 expects provisioning bundles to be preloaded into SRAM. Preload `encrypted_cm_provisioning_bundle_0.bin` to the base of VM0, and `encrypted_dm_provisioning_bundle.bin` to the base of VM1.

If `TFM_DUMMY_PROVISIONING` is disabled and provisioning is required, then BL1_1 will first wait for the TP mode to be set by a debugger (setting the `tp_mode` variable in the current stack frame is easiest). BL1_1 will then wait for provisioning bundles to be loaded to VM0 and VM1 in the same way as when `TFM_DUMMY_PROVISIONING` is enabled, except that it will not automatically perform the reset once each provisioning state is complete. For more details about provisioning flows, see *RSE provisioning*.

RSE communication design

The RSE communication protocol is designed to be a lightweight serialization of the psa_call() API through a combination of in-band MHU (Message Handling Unit) transport and parameter-passing through shared memory.

Message format

To call an RSE service, the client must send a message in-band over the MHU sender link to RSE and wait for a reply message on the MHU receiver (either by polling the MHU or waiting for interrupt). The messages are defined as packed C structs, which are serialized in byte-order over the MHU links.

Messages encoding a psa_call() to RSE take the form:

```
__PACKED_STRUCT serialized_psa_msg_t {
    struct serialized_rse_comms_header_t header;
    __PACKED_UNION {
        struct rse_embed_msg_t embed;
        struct rse_pointer_access_msg_t pointer_access;
    } msg;
};
```

Replies from RSE take the form:

```
__PACKED_STRUCT serialized_psa_reply_t {
    struct serialized_rse_comms_header_t header;
    __PACKED_UNION {
        struct rse_embed_reply_t embed;
        struct rse_pointer_access_reply_t pointer_access;
    } reply;
};
```

All messages (calls and replies), in all protocols carry the following header:

```
__PACKED_STRUCT serialized_rse_comms_header_t {
    uint8_t protocol_ver;
    uint8_t seq_num;
    uint16_t client_id;
};
```

The client_id can be used by the caller to identify different clients at the endpoint for access control purposes. It is combined with an RSE-internal identifier for the endpoint to create the PSA Client ID for the call.

The sequence number, seq_num, is returned in the reply message to allow the client to identify which message is being responded to, since replies may be out-of-order. The sender is free to assign sequence numbers using a scheme of its choice, and it is the sender's responsibility to ensure that two messages with the same sequence number are not in simultaneous flight.

The protocol_ver identifies which message protocol is being used. There are two protocols currently supported, "embed" (protocol_ver=0) and "pointer access" (protocol_ver=1).

Embed protocol

The embed protocol embeds the psa_call iovecs into the message sent over the MHU. It has the following format:

```
__PACKED_STRUCT rse_embed_msg_t {
    psa_handle_t handle;
    uint32_t ctrl_param;
    uint16_t io_size[PSA_MAX_IOVEC];
    uint8_t payload[RSE_COMMS_PAYLOAD_MAX_SIZE];
};
```

The handle is the psa_call handle parameter and the ctrl_param packs the type, in_len and out_len parameters of psa_call into one parameter.

The io_size array contains the sizes of the up to PSA_MAX_IOVEC (4) iovecs, in order, with the invec sizes before the outvec sizes.

The payload array then contains the invec data packed contiguously in order. The length of this parameter is variable, equal to the sum of the invec lengths in io_size. The caller does not need to pad the payload to the maximum size. The maximum payload size for this protocol, RSE_COMMS_PAYLOAD_MAX_SIZE, is a build-time option.

Replies in the embed protocol take the form:

```
__PACKED_STRUCT rse_embed_reply_t {
    int32_t return_val;
    uint16_t out_size[PSA_MAX_IOVEC];
    uint8_t payload[RSE_COMMS_PAYLOAD_MAX_SIZE];
};
```

The return_val is the return value of the psa_call() invocation, the out_size is the (potentially updated) sizes of the outvecs and the payload buffer contains the outvec data serialized contiguously in outvec order.

Pointer access protocol

The pointer access protocol passes the psa_call iovecs as pointers to shared memory using the following MHU message format:

```
__PACKED_STRUCT rse_pointer_access_msg_t {
    psa_handle_t handle;
    uint32_t ctrl_param;
    uint32_t io_sizes[PSA_MAX_IOVEC];
    uint64_t host_ptrs[PSA_MAX_IOVEC];
};
```

The handle, ctrl_param and io_sizes have the same definition as in the embed protocol.

The host_ptrs point to each of the up to PSA_MAX_IOVEC iovec buffers in host system memory. It is the caller's responsibility to write the invec data to the invec pointers before sending the message, and ensure that the memory pointed-to remains valid for the duration of the call.

The reply message has the form:

```
__PACKED_STRUCT rse_pointer_access_reply_t {
    int32_t return_val;
    uint32_t out_size[PSA_MAX_IOVEC];
};
```

The return_val and out_size have the same meaning as in the embed protocol.

RSE writes the outvec data to the pointers supplied in the call message prior to sending the MHU reply message, so no further payload is sent in the reply message.

Implementation structure

The RSE side of the communication implementation is located in platform/ext/target/arm/rse/common/rse_comms. The implementation is structured as follows:

- rse_comms.c: Implements the TF-M RPC layer using RSE comms implementation.
- rse_comms_hal.c: Abstracts MHU message sending and receiving.
- rse_comms_protocol.c: The common part of the RSE comms protocol.
- rse_comms_protocol_embed.c: The embed RSE comms protocol.
- rse_comms_protocol_protocol_access.c: The pointer access RSE comms protocol.
- rse_comms_atu.c: Allocates and frees ATU regions for host pointer access.
- rse_comms_permissions_hal.c: Checks service access permissions and pointer validity.

A reference implementation of the client side of the RSE comms is available in the Trusted Firmware-A repository.

RSE key management

The RSE has a set of hardware components that act together to prevent hardware protected keys being disclosed to compromised software. This chain involves the LifeCycle Manager "LCM", the Key Management Unit "KMU", and the cryptographic accelerator in the integration layer of the RSE (which in reference RSE builds is a CryptoCell-3XX series accelerator).

Hardware protected keys (henceforth "Hardware Keys") are stored in the LCM, in protected keyslots. On production-mode "PCI" chips these slots are write-only. They are protected during provisioning from disclosure by Secure Provisioning "SP" mode disabling debug, and the encryption with either the RTL key or the CM provisioning key. On boot, the LCM checks the integrity of the stored hardware keys by comparing the amount of zeroes with a stored zero-count which, because of the property of the OTP that does not allow 1-bits to be set to 0-bits, allows detection of any changes in the keys. If the check succeeds it exports the keys to the KMU over a private memory bus (one that is not accessible programs running on the RSE' CPU or other controllers on the main bus such as the DMA).

The KMU has between 2 and 32 hardware keyslots, which are 256 bit (32 byte) in size. Between 0 and 8 of these are hardware keyslots, and correspond to the hardware keyslots in the LCM. These slots are filled by the key export from the LCM, and have their permissions preconfigured so they are not readable or writable by software. Software KMU keyslots (keyslots that are not defined in hardware as hardware keyslots) can be read and written, and then subsequently locked to prevent reading and writing, so that they behave like hardware keyslots. This allows trusted immutable firmware to set up keys for less-trusted later stage code, which can be used but cannot be read. KMU keyslots can be exported over another private bus to the cryptographic accelerator. Hardware keyslots can be invalidated, making them unusable until a hard reset.

The cryptographic accelerator receives the key into a write-only key register, and then can use it to perform cryptographic operations. For the CryptoCell-3XX series, this is limited to AES. As with the previous stages, software cannot retrieve the key, only use it. Note that currently this can only be used with the cc3xx_rom_lib driver, not the cryptocell-312-runtime driver, as the latter requires modification. Note that this path is the only way to use hardware protected keys on the RSE, they cannot be used by software cryptographic libraries that require key material to be accessible by software.

The RSE uses this functionality to allow key derivation (based on NIST SP800-108 with an RSA-based PRF that can utilise the hardware keys as described above) from the HUK and GUK. This allows the derivation of the CCA platform attestation key / delegated attestation root key without allowing access to the GUK directly. As a further security measure, these keys are derived by ROM/OTP code, and the HUK and GUK are then invalidated so they cannot be used by the runtime firmware and BL2.

RSE provisioning

Provisioning theory

The LifeCycle Manager (LCM) controls access to the RSE OTP, and includes a state-machine that controls Lifecycle-state (LCS) transitions. The LCM is derived from the OTP management and state machine subsystems of the CryptoCell-3XX series accelerators, and will be familiar to those who have worked with them.

When the chip hasn't been provisioned, the OTP is blank, which means the LCM is in "Virgin" Test/Production mode "TP mode" state. The first step for provisioning must be to set the LCM to either test-chip mode "TCI" or production-chip mode "PCI". In TCI mode the RTL key is masked to avoid disclosure, several OTP fields are changed from write-only to read-write, to aid in debugging, and debugging is not limited in secure provisioning mode (though the RTL key reads as zero). If `TFM_DUMMY_PROVISIONING` is enabled in cmake (which it is by default) then the chip will be set to TCI mode. If this option is not enabled, execution will pause to allow the setting to be set by a debugger.

Once the TP mode has been set, the chip is then is Chip Manufacturer provisioning state "CM". This mode is intended for the provisioning of the HUK, GUK, CM provisioning key, CM code-encryption key, the root-of-trust public key and the CM config. To provision these fields, The firmware must first receive a provisioning bundle via a debugger. This bundle must be placed at the start of VM0 + an offset the size of the OTP DMA ICS (usually 0x400). This bundle contains the keys and also code to perform the provisioning such as a driver for the LCM, and a function to randomly generate the HUK via the CryptoCell TRNG. The chip must then enter secure provisioning mode by setting the SP_ENABLE register. This causes a reset (but does not clear the RSE SRAMs), and allows access to the RTL key by exporting it to the KMU, though in secure provisioning mode the ability to debug the RSE is disabled, to prevent disclosure of the decrypted provisioning bundle values. The RSE will then decrypt and authenticate the bundle using the RTL key. Under TCI mode the RTL key is zeroed, the bundle generation tool must use a zeroed key to encrypt and sign the bundle. Once the CM provisioning bundle has been unpacked, the RSE will execute the code which will provision the CM provisioning data into OTP. The RSE must be cold-reset, which will disable secure provisioning mode. If `TFM_DUMMY_PROVISIONING` is enabled the reset will happen automatically, else the external provisioning device should read the provisioning state from the GPIO/PSI (which is set via the `rse_sysctrl` register) and perform the reset.

After the cold reset, the RSE will automatically transition to Device Manufacturer provisioning state "DM" as the LCM hardware state-machine reads the values of the cm_config_1 and cm_config_2 fields as non-zero. This state is designed to provision the DM provisioning key, the DM code-encryption key and the DM config. The procedure follows the same steps as the CM provisioning flow, with the exception that the bundle will now be encrypted and signed using the CM provisioning key and must be placed at the base of VM1. As before, once the provisioning bundle has been unpacked/run, the RSE must either be cold-reset or will perform this automatically.

After the cold reset, the device will now be in Secure Enable "SE" mode, due to the dm_config_1 field being non-zero. Debug may be limited based on the hardware DCU mask for SE state. Provisioning will not be run on boot.

Practical RSE provisioning

The RSE buildsystem produces two provisioning bundles (containing both code and data), and then encrypts and signs them with the RTL key to produce `encrypted_cm_provisioning_bundle.bin` and `encrypted_dm_provisioning_bundle.bin`.

Note: Currently encrypted provisioning bundles are not supported due to a lack of AEAD encryption support in the cc312-rom-lib driver. The encrypted_*_provisioning_bundle.bin files should still be used, but note that their contents are not encrypted.

On first boot, the RSE is in Virgin state. If the RSE firmware was built with `TFM_DUMMY_PROVISIONING` enabled then it will automatically set the chip to TCI mode and cold-reset. Production ROM implementations must disable `TFM_DUMMY_PROVISIONING`, which will cause RSE to loop in the ROM until either TCI or PCI mode is set with a debugger. It is possible to set the TP mode in the LCS registers directly, however it may be easier to set the `tp_mode` variable in the frame where RSE is looping, at which point the loop will exit and the TP mode will be set by the ROM code.

On non-virgin boot in CM lifecycle state, RSE checks the start of VM0 for the magic constant `0xC0DEFEED`, which is required to be the first word in the CM provisioning bundle. There is also a second check for a constant at the end of the bundle to ensure the bundle has finished writing. The RSE will perform this check in a loop until a bundle is found.

This procedure is repeated for DM LCS, except that the magic constant is `0xBEEFFEED` and the bundle must be loaded to the base of VM1. Note that the size of RSE memory may vary depending on implementation, so the load address of the DM bundle may change.

In production systems it is intended that these bundles are loaded by a debugger, but for development systems it may be too onerous to perform this procedure, particularly if the system is one that has ephemeral OTP such as an FVP. The preferred solution is to preload to the provisioning bundles into VM0 and VM1 as part of the image loading, which is supported on FVPs but may not be on other systems. An alternative solution is to perform provisioning manually once, and then to save the state of the OTP in SE LCS and then preload that on subsequent boots.

RSE provisioning GPIO signalling

The state of the RSE ROM boot/provisioning flow is signalled outside of the RSE subsystem via the GPIOs as part of the Persistent State Interface (PSI). The PSI signals the lifecycle state as a hardware signal, but additionally the software can signal over the PSI by setting the `rse_sysctrl` register.

The boot state is encoded in the lowest 4 bits of the `rse_sysctrl` register, and has meaning as follows:

Signal	State
0x0	RSE cold boot default
0x1	Virgin chip idle, ready to set PCI/TCI mode
0x2	CM LCS idle, waiting for CM provisioning bundle
0x3	RMA LCS idle
0x4	CM secure provisioning started, secure provisioning mode enabled
0x5	CM secure provisioning failed due to bundle authentication error
0x6	CM secure provisioning failed due to other error
0x7	CM secure provisioning succeeded
0x8	DM LCS idle, waiting for CM provisioning bundle
0x9	DM secure provisioning started, secure provisioning mode enabled
0xa	DM secure provisioning failed due to bundle authentication error
0xb	DM secure provisioning failed due to other error
0xc	DM secure provisioning succeeded
0xd	SE LCS standard boot

RSE to RSE routing

In multi-RSE implementations, RSE uses a routing protocol to allow messages to be sent from any RSE to any other RSE, regardless of if there is a physical link between them. This may involve an intermediate RSE forwarding the message.

A routing table is used to indicate which is the next-hop for any message. This table is indexed by the destination RSE and contains an index indicating which hardware link should be used to send the message.

The table is located in OTP, which allows for it to be provisioned once the chip topology is known. This also allows BL1_2 code to be identical across all RSEs in a system.

The routing table is automatically constructed by the build system. Each platform must specify a .tgf (trivial graph format) file which describes the system topology, which is then used by the provisioning scripts to calculate the next-hop for each RSE to every other RSE. This generates individual provisioning bundles for each RSE containing their routing tables.

RSE System Topology Map

The system topology map is described as a .tgf file. The MULTI_RSE_TOPOLOGY_FILE cmake variable configures which system topology map tgf file will be used by the build system. This variable should be set in RSE subplatform default config for any subplatform which contains multiple RSEs.

Documentation of the TGF format can be found at https://en.wikipedia.org/wiki/Trivial_Graph_Format#Format.

In the system topology map, each node represents an RSE. The Label portion of the node is optional, but it is recommended to use it to describe which RSE the node represents. Each edge represents a physical link between the corresponding RSEs. The edges are directional, with the first index representing the RSE whose link IDs are described by the edge (henceforth referred to as the Local RSE) and the second index representing the RSE which is being communicated with (henceforth referred to as the Remote RSE).

An example graph node line is:

```
0 Primary RSE
```

Because knowing that a link exists does not encode which link should be used, the link ID is encoded as the name of the edge. The format of this name must be "Send <n> Receive <m>" where <n> is the index of the link which is used by the Local RSE to send to the Remote RSE, and <m> is the index of the link which is used by the Local RSE to receive from the Remote RSE.

An example graph edge line is:

```
0 1 Send 1 Receive 2
```

In this example, the Local RSE has index 0 and the Remote RSE has index 1. If the Local RSE wants to send a message to the Remote RSE, it should send the message via the link with index 1. If the Local RSE wants to listen for messages from the Remote RSE, it should listen on the link with index 2.

The following is an example topology map for a system with 4 RSEs, which are all connected to each other:

```
0 RSE 0
1 RSE 1
2 RSE 2
3 RSE 3

0 1 Send 1 Receive 1
0 2 Send 2 Receive 2
0 3 Send 3 Receive 3

1 0 Send 0 Receive 0
1 2 Send 2 Receive 2
1 3 Send 3 Receive 3

2 0 Send 0 Receive 0
2 1 Send 1 Receive 1
2 3 Send 3 Receive 3

3 0 Send 0 Receive 0
3 1 Send 1 Receive 1
3 2 Send 2 Receive 2
```

The following is a "hub-and-spoke" system topology with 5 RSEs:

```
0 Hub
1 Spoke 0
2 Spoke 1
3 Spoke 2
4 Spoke 3

0 1 Send 0 Receive 0
0 2 Send 1 Receive 1
0 3 Send 2 Receive 2
0 4 Send 3 Receive 3

1 0 Send 0 Receive 0
2 0 Send 0 Receive 0
3 0 Send 0 Receive 0
4 0 Send 0 Receive 0
```

RSE Routing Table

The RSE routing table is provisioned into OTP. It is provisioned as two distinct tables, as "send" table and a "receive" table. Each table entry is a uint32_t, which is used as an index into the link table in order to decide which link to send/receive on. The amount of elements in the routing table is the same as the number of RSEs in the system.

In order to find the link that should be used, the destination RSE should be used as an index in the correct routing table. The value in the routing table should then be used to select the link.

The size of the routing tables is 4 bytes * 2 * the number of RSEs in the system.

The value in the table which corresponds to the current RSE must be 0. It is invalid to send a message to the same RSE that is sending the message. 0 is also a valid link index.

Each RSE in the system is provisioned with different routing tables. The routing tables for each RSE are generated by running a shortest-path algorithm on the graph between the RSE whose table is being generated and every other RSE.

The link table for RSE 2 in the first example above (fully connected) is:

```
Send    = {0, 1, 0, 3}
Receive = {0, 1, 0, 3}
```

The link table for RSE 1 in the Second example above (hub-and-spoke) is:

```
Send    = {0, 0, 0, 0}
Receive = {0, 0, 0, 0}
```

RSE ROM Releases

RSE ROM Release 2024-04-08

Contents

Features

- A ROM bootloader for RSE that supports chainloading a second immutable boot stage (BL1-2) from OTP.
- Integrity protection of the BL1-2 boot image using a provisioned SHA-256 hash of the image.
- Hardware-accelerated crypto operations using the built-in CryptoCell-3xx ROM driver.
- Support for hardware lifecycle management and provisioning using code and values loaded through a debugger.
- A DMA Initial Command Sequence placed at offset 0x1F000 in the ROM.
- Support for secure provisioning, using AEAD encryption with keys derived from the RTL key.
- Allows for setup of SAM configuration with provisioned values before CPU is started via the DMA ICS.
- Uses TRAM for BL1_1 data sections.

Known errata

- The build instructions for this release initially contained an incorrect address for the DMA ICS concatenation with the ROM code. The build instructions in this document have been updated and are now correct. The correct address is 0x1F000, instead of 0x1E000. Some distributed ROM binaries have also been impacted, and hence will not boot.

  If a ROM binary does not boot, it can be checked for this error using the command:

  ```
  xxd -s 0x1F000 rom.bin
  ```

  If there is no output, or the output is all zero-words, then the ROM binary is affected by the issue. It is possible to fix a ROM binary affected by the issue by running the following command:

  ```
  dd if=rom.bin bs=1k skip=120 seek=124 count=4 of=rom.bin
  ```

TF-M version

The RSE ROM image should be generated from TF-M commit hash:

```
2ca8c58dc2feddc93b87585ffc07e4d169f54278
```

Tested Environment

- arm-none-eabi-gcc –version | head -n 1

```
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10.3-2021.10) 10.3.1 20210824 (release)
```

- python3 –version

```
Python 3.11.6
```

- python3 -m pip list

```
Package                   Version
------------------------- ---------
attrs                     23.2.0
cbor2                     5.6.2
cffi                      1.16.0
click                     8.1.7
cryptography              42.0.5
ecdsa                     0.18.0
imgtool                   2.0.0
intelhex                  2.3.0
Jinja2                    3.1.3
jsonschema                4.21.1
jsonschema-specifications 2023.12.1
kconfiglib                14.1.0
MarkupSafe                2.1.5
networkx                  3.3
pip                       23.2
pyasn1                    0.6.0
pycparser                 2.22
pyhsslms                  2.0.0
PyYAML                    6.0.1
referencing               0.34.0
rpds-py                   0.18.0
setuptools                68.1.2
six                       1.16.0
```

Build command

The RSE ROM image should be generated with the following build commands:

```
python3 -m venv ./venv
source ./venv/bin/activate

python3 -m pip install -r ./tools/requirements.txt

cmake -S . -B build -DTFM_PLATFORM=arm/rse/tc \
    -DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake \
    -DCMAKE_BUILD_TYPE=minsizerel \
    -DRSE_ENABLE_TRAM=ON \
    -DRSE_OTP_TRNG=ON \
    -DTFM_DUMMY_PROVISIONING=OFF \
    -DRSE_XIP=ON \
    -DTFM_PARTITION_DPE=OFF

cmake --build build -- install

srec_cat build/bin/bl1_1.bin  -Binary -offset 0x0 \
    build/bin/rom_dma_ics.bin -Binary -offset 0x1F000 \
    -o rse_rom_2024-04-08.bin -Binary

truncate --size 131072 rse_rom_2024-04-08.bin
```

RSE also includes the following extra partitions:

- Authenticated Debug Access Control (ADAC)
- Measured boot partition
- Delegated attestation partition
- DICE Protection Environment partition

7.2 ArmChina platforms

7.2.1 Arm China Alcor with Cortex-M52 Example Subsystem for MPS3 (AN557)

Introduction

Alcor is an ArmChina reference subsystem for secure System on Chips containing an Armv8.1-M Cortex-M52 processor. It is an MPS3 based platform with the usual MPS3 peripherals.

This platform port supports all TF-M regression tests (Secure and Non-secure) with Isolation Level 1 and 2.

Note: For Armclang compiler v6.18 or later version is required.

Building TF-M

Follow the instructions in Building instructions.

Build instructions with platform name: armchina/mps3/alcor/an557

```
-DTFM_PLATFORM=armchina/mps3/alcor/an557
```

Note: Provisioning bundles can be generated with the `-DPLATFORM_DEFAULT_PROVISIONING=OFF` flag. The provisioning bundle binary will be generated and it's going to contain the provisioning code and provisioning values.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` and `-DTFM_DUMMY_PROVISIONING=ON` then the keys in the `tf-m/platform/ext/target/armchina/mps3/common/provisioning/provisioning_config.cmake` and the default MCUBoot signing keys will be used for provisioning.

If -DPLATFORM_DEFAULT_PROVISIONING=OFF and -DTFM_DUMMY_PROVISIONING=OFF are set then unique values can be used for provisioning. The keys and seeds can be changed by passing the new values to the build command, or by setting the -DPROVISIONING_KEYS_CONFIG flag to a .cmake file that contains the keys. An example config cmake file can be seen at tf-m/platform/ext/target/armchina/mps3/common/provisioning/provisioning_config.cmake. Otherwise new random values are going to be generated and used. For the image signing the ${MCUBOOT_KEY_S} and ${MCUBOOT_KEY_NS} will be used. These variables should point to .pem files that contain the code signing private keys. The public keys are going to be generated from these private keys and will be used for provisioning. The hash of the public key is going to be written into the provisioning_data.c automatically.

If -DMCUBOOT_GENERATE_SIGNING_KEYPAIR=ON is set then a new mcuboot signing public and private keypair is going to be generated and it's going to be used to sign the S and NS binaries.

The new generated keypair can be found in the <build dir>/bin folder or in the <install directory>/image_signing/keys after installation. The generated provisioning_data.c file can be found at <build directory>/platform/target/provisioning/provisioning_data.c

Note: The provisioning bundle generation depends on pyelftools that's have to be installed:

```
pip3 install pyelftools
```

To run the example code on Arm China Alcor with Cortex-M52 Example Subsystem for MPS3 (AN557)

To run BL2 bootloader, TF-M example application and tests in the MPS3 board, it is required to have AN557 image in the MPS3 board SD card. The image should be located in <MPS3 device name>/MB/HBI<BoardNumberBoardrevision>/AN557

The MPS3 board tested is HBI0309C.

1. Copy bl2.bin and tfm_s_ns_signed.bin files from build dir to <MPS3 device name>/SOFTWARE/
2. Rename tfm_s_ns_signed.bin to tfm.bin (Filename should not be longer than 8 charachters.)
3. Open <MPS3 device name>/MB/HBI0309C/AN557/images.txt
4. Update the images.txt file as follows:

```
TITLE: Arm MPS3 FPGA prototyping board Images Configuration File

[IMAGES]
TOTALIMAGES: 2                      ;Number of Images (Max: 32)

IMAGE0UPDATE: AUTO                  ;Image Update:NONE/AUTO/FORCE
IMAGE0ADDRESS: 0x00000000           ;Please select the required executable program
IMAGE0FILE: \SOFTWARE\bl2.bin
IMAGE1UPDATE: FORCEQSPI
IMAGE1ADDRESS: 0x00000000
IMAGE1FILE: \SOFTWARE\tfm.bin
```

5. Close <MPS3 device name>/MB/HBI0309C/AN557/images.txt
6. Unmount/eject the <MPS3 device name> unit
7. Reset the board to execute the TF-M example application

8. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[INF] Image index: 1, Swap type: none
[INF] Image index: 0, Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT␣
↪SECURE
[WRN] This device was provisioned with dummy keys. This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v2.0.0
```

Note: Some of the messages above are only visible when `CMAKE_BUILD_TYPE` is set to `Debug`.

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x10022400` address by copying `encrypted_provisioning_bundle.bin` and renaming it to `prv.bin`, then extending the images.txt with:

```
IMAGE2UPDATE: AUTO
IMAGE2ADDRESS: 0x00022400
IMAGE2FILE: \SOFTWARE\prv.bin
```

7.3 Cypress platform

7.3.1 Cypress PSoC64

Cypress PSoC64 Specifics

Prerequisites

PSoC64 must first be provisioned with SecureBoot firmware and a provisioning packet containing policy and secure keys. Please refer to the guide at https://www.cypress.com/documentation/software-and-drivers/psoc-64-secure-mcu-secure-boot-sdk-user-guide Use the following policy file for provisioning and signing: policy_multi_CM0_CM4_tfm.json

Please make sure you have all required software installed as explained in the TF-M getting started.

Please also make sure that all the source code are fetched by following *general building instruction*.

Install CySecureTools or update it to the newest version (2.0.0 at the time of writing)

```
pip3 install --upgrade --force-reinstall cysecuretools
```

PyOCD is installed by CySecureTools as a dependency.

For more details please refer to CySecureTools page.

Install OpenOCD with PSoC6 support. Download the latest revision 4.1.0 or newer from: https://github.com/cypresssemiconductorco/openocd/releases

Lastly, the PSoC64 board KitProg firmware needs to be 2.00.744 or greater. Please use Cypress Modus Toolbox or CyProgrammer to update it if needed.

Building Multi-Core TF-M on Cypress PSoC64

Configuring the build

The build configuration for TF-M is provided to the build system using command line arguments. Required arguments are noted below.

- - -DTFM_PLATFORM=cypress/psoc64
  - Specifies target platform name `psoc64`
- - -DTFM_TOOLCHAIN_FILE=<path to toolchain file>
  - Specifies the compiler toolchain The possible values are:
    - `<TFM root dir>/toolchain_ARMCLANG.cmake`
    - `<TFM root dir>/toolchain_GNUARM.cmake`
    - `<TFM root dir>/toolchain_IARARM.cmake`

see *Build configuration* for more information.

Build Instructions

The following instructions build multi-core TF-M without regression test suites in Isolation Level 1 on Linux. Both the compiler and the debugging type can be changed to other configurations listed above.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

mkdir <build folder>
pushd <build folder>
cmake -DTFM_PLATFORM=cypress/psoc64 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      ../
popd
cmake --build <build folder> -- -j VERBOSE=1
```

The following instructions build multi-core TF-M with regression test suites in Isolation Level 1 on Linux. Both the compiler and the debugging type can be changed to other configurations listed above.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

mkdir <build folder>
pushd <build folder>
cmake -DTFM_PLATFORM=cypress/psoc64 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTEST_S=ON -DTEST_NS=ON \
      ../
popd
cmake --build <build folder> -- -j VERBOSE=1
```

The following instructions build multi-core TF-M with PSA API test suite for the attestation service in Isolation Level 1 on Linux. Both the compiler and the debugging type can be changed to other configurations listed above.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

mkdir <build folder>
pushd <build folder>
cmake -DTFM_PLATFORM=cypress/psoc64 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTEST_PSA_API=INITIAL_ATTESTATION \
      ../
popd
cmake --build <build folder> -- -j VERBOSE=1
```

The following instructions build multi-core TF-M without regression test suites in Isolation Level 2 on Linux. Both the compiler and the debugging type can be changed to other configurations listed above.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

mkdir <build folder>
pushd <build folder>
cmake -DTFM_PLATFORM=cypress/psoc64 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_ISOLATION_LEVEL=2 \
      ../
popd
cmake --build <build folder> -- -j VERBOSE=1
```

The following instructions build multi-core TF-M with regression test suites in Isolation Level 2 on Linux. Both the compiler and the debugging type can be changed to other configurations listed above.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

mkdir <build folder>
pushd <build folder>
cmake -DTFM_PLATFORM=cypress/psoc64 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_ISOLATION_LEVEL=2 \
```

(continues on next page)

(continued from previous page)

```
      -DTEST_S=ON -DTEST_NS=ON \
      ../
popd
cmake --build <build folder> -- -j VERBOSE=1
```

The following instructions build multi-core TF-M with PSA API test suite for the protected storage service in Isolation Level 2 on Linux. Both the compiler and the debugging type can be changed to other configurations listed above.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

mkdir <build folder>
pushd <build folder>
cmake -DTFM_PLATFORM=cypress/psoc64 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_ISOLATION_LEVEL=2 \
      -DTEST_PSA_API=PROTECTED_STORAGE \
      ../
popd
cmake --build <build folder> -- -j VERBOSE=1
```

Signing

Signing keys

Copy secure keys used in the board provisioning process to platform/ext/target/cypress/psoc64/security/keys:

The keys included in the repository are for reference and development only. DO NOT USE THESE KEYS IN ANY ACTUAL DEPLOYMENT!

Note: provisioned board in SECURE state is required, otherwise refer to Cypress documentation for details on the provisioning process. https://www.cypress.com/documentation/software-and-drivers/psoc-64-secure-mcu-secure-boot-sdk-user-guide

If the board was previously provisioned with signing keys and policy, copy secure signing keys used in the board provisioning process to platform/ext/target/cypress/psoc64/security/keys:

TFM_S_KEY.json
Private OEM key for signing CM0P image

TFM_S_KEY_PRIV.pem
Private OEM key for signing CM0P image in PEM format

TFM_NS_KEY.json
Private OEM key for signing CM4 image

TFM_NS_KEY_PRIV.pem
Private OEM key for signing CM4 image in PEM format

Alternatively, you can generate a new set of signing keys using cysecuretools create-keys command and provision the keys to the board, if the previously provisioned policy allows board's re-provisioning.

Initialize cysecuretools environment:

```
cd platform/ext/target/cypress/psoc64/security
cysecuretools -t cy8ckit-064s0s2-4343w init
```

Generate a new set of keys:

```
cysecuretools -t cy8ckit-064s0s2-4343w -p policy/policy_multi_CM0_CM4_tfm.json create-
↪keys
```

Re-provision the new keys to the board:

```
cysecuretools -t cy8ckit-064s0s2-4343w -p policy/policy_multi_CM0_CM4_tfm.json re-
↪provision-device
```

Signing the images

Sign the images using CySecureTools CLI tool. Note: the tool overwrites unsigned file with a signed one, it also creates an unsigned copy <filename>_unsigned.hex.

SPE image:

```
cysecuretools \
--policy platform/ext/target/cypress/psoc64/security/policy/policy_multi_CM0_CM4_tfm.
↪json \
--target cy8ckit-064s0s2-4343w \
sign-image \
--hex <build folder>/bin/tfm_s.hex \
--image-type BOOT \
--image-id 1
```

NSPE image:

```
cysecuretools \
--policy platform/ext/target/cypress/psoc64/security/policy/policy_multi_CM0_CM4_tfm.
↪json \
--target cy8ckit-064s0s2-4343w \
sign-image \
--hex <build folder>/bin/tfm_ns.hex \
--image-type BOOT \
--image-id 16
```

- CySecureTools sign-image overwrites unsigned file with a signed one, also it creates an unsigned copy _unsigned.hex.
- image-type option: "–image-type BOOT" creates a signed hex file with offsets for the primary image slot. Use "–image-type UPGRADE" if you want to create an image for the secondary "upgrade" slot. When booting, CyBootloader will validate image in the secondary slot and copy it to the primary boot slot.
- image-id option: Each image has its own ID. By default, SPE image running on CM0P core has ID=1, NSPE image running on CM4 core has ID=16. Refer to the policy file for the actual ID's.

Programming the Device

After building and signing, the TFM images must be programmed into flash memory on the PSoC64 device. There are three methods to program it.

DAPLink mode

Using KitProg3 mode button, switch it to DAPLink mode. Mode LED should start blinking rapidly and depending on the host computer settings DAPLINK will be mounted as a media storage device. Otherwise, mount it manually.

Copy tfm hex files one by one to the DAPLINK device:

```
cp <build folder>/bin/tfm_ns.hex <mount point>/DAPLINK/; sync
cp <build folder>/bin/tfm_s.hex <mount point>/DAPLINK/; sync
```

OpenOCD

Using KitProg3 mode button, switch to KitProg3 CMSIS-DAP BULK. Status LED should be ON and not blinking for CMSIS-DAP BULK.

Before programming the images, erase PS partition if needed. This will clear all data and force PS to reformat partition.

```
${OPENOCD_PATH}/bin/openocd \
        -s ${OPENOCD_PATH}/scripts \
        -f interface/kitprog3.cfg \
        -f target/psoc6_2m_secure.cfg \
        -c "init; reset init" \
        -c "flash erase_address 0x101c0000 0x10000" \
        -c "shutdown"
```

Note that the `0x101C0000` in the command above must match the PS start address of the secure primary image specified in the file:

> platform/ext/target/cypress/psoc64/partition/flash_layout.h

so be sure to change it if you change that file.

To program the signed tfm_s and tfm_ns images to the device with openocd (assuming OPENOCD_PATH is pointing at the openocd installation directory) run the following commands:

```
OPENOCD_PATH=<cyprogrammer dir>/openocd
BUILD_DIR=<build folder>

${OPENOCD_PATH}/bin/openocd \
        -s ${OPENOCD_PATH}/scripts \
        -f interface/kitprog3.cfg \
        -f target/psoc6_2m_secure.cfg \
        -c "init; reset init" \
        -c "flash write_image erase ${BUILD_DIR}/bin/tfm_s.hex" \
        -c "shutdown"

${OPENOCD_PATH}/bin/openocd \
        -s ${OPENOCD_PATH}/scripts \
```

(continues on next page)

(continued from previous page)

```
        -f interface/kitprog3.cfg \
        -f target/psoc6_2m_secure.cfg \
        -c "init; reset init" \
        -c "flash write_image erase ${BUILD_DIR}/bin/tfm_ns.hex" \
        -c "reset run"
```

PyOCD

PyOCD is installed by CySecureTools automatically. It can be used to program TFM images into the board.

Using KitProg3 mode button, switch to KitProg3 DAPLink mode. Mode LED should start blinking rapidly.

Optionally, erase PS partition:

```
pyocd erase -b CY8CKIT-064S0S2-4343W -s 0x101c0000+0x10000
```

To program the signed tfm_s and tfm_ns images to the device with pyocd run the following commands:

```
pyocd flash -b CY8CKIT-064S0S2-4343W ${BUILD_DIR}/bin/tfm_s.hex

pyocd flash -b CY8CKIT-064S0S2-4343W ${BUILD_DIR}/bin/tfm_ns.hex
```

Provisioning device certificates

1. If not done yet, change to the psoc64 security directory and initialize cysecuretools environment:

```
cd platform/ext/target/cypress/psoc64/security
cysecuretools -t cy8ckit-064s0s2-4343w init
```

2. Create and copy rootCA files to “certificates” directory next to the policy directory (please refer to documentation of the used OS)
3. Switch the board in DAPLink mode by pressing the mode button or by issuing the following fw-loader command (fw-loader comes with Modus ToolBox software). Mode LED should be slowly blinking:

```
fw-loader --mode kp3-daplink
```

4. Run reprov_helper.py. If running the script with default parameters, the script can be run as is:

```
 python3 reprov_helper.py

Otherwise, run it with --help parameter to get the full list of options.
```

5. Confirm selected options. When prompted for a serial number, enter the board unique serial number (digits only, e.g. 00183).
6. Script will ask if you want to create new signing keys. Answer Yes to generate new signing keys in the keys directory, or No to retain and use the existing keys. After re-provisioning, from now on any images for this board will have to be signed with these keys.
7. The script will erase user images. Then the script will read device public key and create device certificates based on the board serial number, root certificate and the device public key.

Provisioning Amazon Web Services root and device certificates

Device running Amazon FreeRTOS and using Amazon Web Services (AWS) requires AWS root and unique device certificate signed with the device key for authentication. These certificates have to be provisioned into device. The process is the following:

1. If not done yet, change to the psoc64 security directory and initialize cysecuretools environment:

```
cd platform/ext/target/cypress/psoc64/security
cysecuretools -t cy8ckit-064s0s2-4343w init
```

2. Obtain and copy rootCA files to "certificates" directory next to the policy directory (please refer to AWS documentation)

3. Switch the board in DAPLink mode by pressing the mode button or by issuing the following fw-loader command (fw-loader comes with Modus ToolBox software). Mode LED should be slowly blinking:

```
fw-loader --mode kp3-daplink
```

4. Run reprov_helper.py. If running the script with default parameters, the script can be run as is:

```
 python3 reprov_helper.py

Otherwise, run it with --help parameter to get the full list of options.
```

5. Confirm selected options. When prompted for a serial number, enter the board unique serial number (digits only, e.g. 00183).

6. Script will ask if you want to create new signing keys. Answer Yes to generate new signing keys in the keys directory, or No to retain and use the existing keys. After re-provisioning, from now on any images for this board will have to be signed with these keys.

7. The script will erase user images. Then the script will read device public key and create device certificates based on the board serial number, root certificate and the device public key.

Core Library

Overview

The Core Library provides basic types and utilities that can be used between different devices. This allows different libraries to share common items between themselves to avoid reimplementation and promote consistency.

Features

- Common result type (cy_rslt_t) for reporting errors or status
- Common utility MACROs for
  - CY_ASSERT: Verifies a value and halts if invalid (if not NDEBUG)
  - CY_HALT: Halts the application
  - CY_UNUSED_PARAMETER: Avoid warnings if a function argument isn't used
  - CY_LO8: Gets the lower 8 bits of a 16-bit value
  - CY_HI8: Gets the upper 8 bits of a 16-bit value
  - CY_LO16: Gets the lower 16 bits of a 32-bit value
  - CY_HI16: Gets the upper 16 bits of a 32-bit value
  - CY_SWAP_ENDIAN16: Swaps the byte ordering of a 16-bit value
  - CY_SWAP_ENDIAN32: Swaps the byte ordering of a 32-bit value
  - CY_SWAP_ENDIAN64: Swaps the byte ordering of a 64-bit value
  - CY_GET_REG8: Reads the 8-bit value from the specified address
  - CY_SET_REG8: Writes an 8-bit value to the specified address
  - CY_GET_REG16: Reads the 16-bit value from the specified address
  - CY_SET_REG16: Writes the 16-bit value to the specified address
  - CY_GET_REG24: Reads the 24-bit value from the specified address
  - CY_SET_REG24: Writes the 24-bit value to the specified address
  - CY_GET_REG32: Reads the 32-bit value from the specified register
  - CY_SET_REG32: Writes the 32-bit value to the specified register
  - _CLR_SET_FLD32U: The macro for setting a register with a name field and value for providing get-clear-modify-write operations
  - CY_REG32_CLR_SET: Uses _CLR_SET_FLD32U macro for providing get-clear-modify-write operations with a name field and value and writes a resulting value to the 32-bit register
  - _CLR_SET_FLD16U: The macro for setting a 16-bit register with a name field and value for providing get-clear-modify-write operations
  - CY_REG16_CLR_SET: Uses _CLR_SET_FLD16U macro for providing get-clear-modify-write operations with a name field and value and writes a resulting value to the 16-bit register
  - _CLR_SET_FLD8U: The macro for setting a 8-bit register with a name field and value for providing get-clear-modify-write operations
  - CY_REG8_CLR_SET: Uses _CLR_SET_FLD8U macro for providing get-clear-modify-write operations with a name field and value and writes a resulting value to the 8-bit register
  - _BOOL2FLD: Returns a field mask if the value is not false
  - _FLD2BOOL: Returns true, if the value includes the field mask
  - CY_SYSLIB_DIV_ROUND: Calculates a / b with rounding to the nearest integer, a and b must have the same sign.

  - CY_SYSLIB_DIV_ROUNDUP: Calculates a / b with rounding up if remainder != 0, both a and b must be positive.

- MACROs to create cross compiler compatible code. Use the CY_NOINIT, CY_SECTION, CY_UNUSED, CY_ALIGN attributes at the first place of declaration/definition. For example: CY_NOINIT uint32_t noinitVar;
  - CY_NOINIT
  - CY_SECTION
  - CY_UNUSED
  - CY_NOINLINE
  - CY_ALIGN
  - CY_RAMFUNC_BEGIN
  - CY_RAMFUNC_END

More information

- API Reference Guide
- Cypress Semiconductor, an Infineon Technologies Company
- Cypress Semiconductor GitHub
- ModusToolbox

Core Library Release Notes

The Core Library provides basic types and utilities that can be used between different devices. This allows different libraries to share common items between themselves to avoid reimplementation and promote consistency.

What's Included?

- Common result type for reporting errors or status
- Common utility MACROs for
  - CY_ASSERT: Verifies a value and halts if invalid (if not NDEBUG)
  - CY_HALT: Halts the application
  - CY_UNUSED_PARAMETER: Avoid warnings if a function argument isn't used
  - CY_LO8: Gets the lower 8 bits of a 16-bit value
  - CY_HI8: Gets the upper 8 bits of a 16-bit value
  - CY_LO16: Gets the lower 16 bits of a 32-bit value
  - CY_HI16: Gets the upper 16 bits of a 32-bit value
  - CY_SWAP_ENDIAN16: Swaps the byte ordering of a 16-bit value
  - CY_SWAP_ENDIAN32: Swaps the byte ordering of a 32-bit value

  - CY_SWAP_ENDIAN64: Swaps the byte ordering of a 64-bit value
  - CY_GET_REG8: Reads the 8-bit value from the specified address
  - CY_SET_REG8: Writes an 8-bit value to the specified address
  - CY_GET_REG16: Reads the 16-bit value from the specified address
  - CY_SET_REG16: Writes the 16-bit value to the specified address
  - CY_GET_REG24: Reads the 24-bit value from the specified address
  - CY_SET_REG24: Writes the 24-bit value to the specified address
  - CY_GET_REG32: Reads the 32-bit value from the specified register
  - CY_SET_REG32: Writes the 32-bit value to the specified register
  - _CLR_SET_FLD32U: The macro for setting a register with a name field and value for providing get-clear-modify-write operations
  - CY_REG32_CLR_SET: Uses _CLR_SET_FLD32U macro for providing get-clear-modify-write operations with a name field and value and writes a resulting value to the 32-bit register
  - _CLR_SET_FLD16U: The macro for setting a 16-bit register with a name field and value for providing get-clear-modify-write operations
  - CY_REG16_CLR_SET: Uses _CLR_SET_FLD16U macro for providing get-clear-modify-write operations with a name field and value and writes a resulting value to the 16-bit register
  - _CLR_SET_FLD8U: The macro for setting a 8-bit register with a name field and value for providing get-clear-modify-write operations
  - CY_REG8_CLR_SET: Uses _CLR_SET_FLD8U macro for providing get-clear-modify-write operations with a name field and value and writes a resulting value to the 8-bit register
  - _BOOL2FLD: Returns a field mask if the value is not false
  - _FLD2BOOL: Returns true, if the value includes the field mask
  - CY_SYSLIB_DIV_ROUND: Calculates a / b with rounding to the nearest integer, a and b must have the same sign.
  - CY_SYSLIB_DIV_ROUNDUP: Calculates a / b with rounding up if remainder != 0, both a and b must be positive.

- MACROs to create cross compiler compatible code. Use the CY_NOINIT, CY_SECTION, CY_UNUSED, CY_ALIGN attributes at the first place of declaration/definition. For example: CY_NOINIT uint32_t noinitVar;

  - CY_NOINIT
  - CY_SECTION
  - CY_UNUSED
  - CY_NOINLINE
  - CY_ALIGN
  - CY_RAMFUNC_BEGIN
  - CY_RAMFUNC_END

What Changed?

v1.1.4

- Minor updates for MISRA & documentation #### v1.1.3
- Minor update for documentation & branding #### v1.1.2
- Added new module identifiers to cy_result.h #### v1.1.1
- Fixed MISRA violation #### v1.1.0
- Migrated numerous utility & cross compiler macros from psoc6pdl into here #### v1.0.1
- Added new module IDs in cy_result.h
- Minor updates to documentation #### v1.0.0
- Initial release

Supported Software and Tools

This version of the Core Library was validated for compatibility with the following Software and Tools:

Software and Tools	Version
ModusToolbox Software Environment	2.1
GCC Compiler	9.2
IAR Compiler	8.4
ARM Compiler	6.11

Minimum required ModusToolbox Software Environment: v2.0

More information

Use the following links for more information, as needed:

- API Reference Guide
- Cypress Semiconductor, an Infineon Technologies Company
- Cypress Semiconductor GitHub
- ModusToolbox

7.4 Laird Connectivity platform

7.4.1 Laird Connectivity BL5340

The BL5340 development kit (DVK, 750-03293) is a single-board development kit for the evaluation and development on the Laird Connectivity BL5340 module which contains a Nordic Semiconductor nRF5340 System-on-Chip (SoC).

The BL5340 is a module with a dual-core nRF5340 SoC based on the Arm® Cortex®-M33 architecture, with:

- a full-featured ARM Cortex-M33F core with DSP instructions, FPU, and ARMv8-M Security Extension, running at up to 128 MHz, referred to as the **Application MCU**
- a secondary ARM Cortex-M33 core, with a reduced feature set, running at a fixed 64 MHz, referred to as the **Network MCU**.

The BL5340 Application MCU supports the Armv8m Security Extension.

Documentation

The following links provide useful information about the BL5340

BL5340 website:
https://www.lairdconnect.com/wireless-modules/bluetooth-modules/bluetooth-5-modules/bl5340-series-multi-core-bluetooth-52-802154-nfc-modules

Nordic Semiconductor Infocenter: https://infocenter.nordicsemi.com

Building TF-M on BL5340 Application MCU

To build an S and NS application image for the BL5340 Application MCU run the following commands:

Note: On OS X change `readlink` to `greadlink`, available by running `brew install coreutils`.

```
$ mkdir build && cd build
$ cmake -DTFM_PLATFORM=lairdconnectivity/bl5340_dvk_cpuapp \
        -DTFM_TOOLCHAIN_FILE=../toolchain_GNUARM.cmake \
        -G"Unix Makefiles" ../
$ make install
```

Note: Currently, applications can only be built using GCC (GNU ARM Embedded toolchain).

Note: For BL2 (MCUBoot) logging output to be available, the project needs to be built with Debug configuration (CMAKE_BUILD_TYPE=Debug).

Flashing and debugging with on-board Segger J-Link

The BL5340 DVK is equipped with a Debug IC (Atmel ATSAM3U2C) which provides the following functionality:

- Segger J-Link firmware and desktop tools
- SWD debug for the BL5340 module
- USB CDC ACM Serial Port bridged to the second UART peripheral
- Segger RTT Console

To install the J-Link Software and documentation pack, follow the steps below:

#. Download the appropriate package from the J-Link Software and documentation pack website #. Depending on your platform, install the package or run the installer #. When connecting a J-Link-enabled board such as a BL5340 DVK, a serial port should come up

nRF Command-Line Tools Installation

The nRF Command-line Tools allow you to control your BL5340 module from the command line, including resetting it, erasing or programming the flash memory and more.

To install them, visit nRF Command-Line Tools and select your operating system.

After installing, make sure that `nrfjprog` is somewhere in your executable path to be able to invoke it from anywhere.

BL2, S, and NS application images can be flashed into BL5340 separately or may be merged together into a single binary.

Flashing the BL5340 DVK

To program the flash with a compiled TF-M image (i.e. S, NS or both) after having followed the instructions to install the Segger J-Link Software and the nRF Command-Line Tools, follow the steps below:

Generate Intel hex files from the output binary (bin) files as follows:

```
srec_cat install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/tfm_s_ns_signed.bin -binary␣
↪--offset=0x10000 -o install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/tfm_s_ns_
↪signed.hex -intel
```

- Connect the micro-USB cable to the BL5340 DVK and to your computer
- Erase the flash memory in the BL5340 module:

```
nrfjprog --eraseall -f nrf53
```

- Flash the BL2 and the TF-M image binaries from the sample folder of your choice:

```
nrfjprog --program <sample folder>/install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/
↪bl2.hex -f nrf53 --sectorerase
nrfjprog --program <sample folder>/install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/
↪tfm_s_ns_signed.hex -f nrf53 --sectorerase
```

- Reset and start TF-M:

```
nrfjprog --reset -f nrf53
```

Flashing the BL5340 DVK (Secondary slot in QSPI, with BL2)

To program the flash with a compiled TF-M image (i.e. S, NS or both) after having followed the instructions to install the Segger J-Link Software and the nRF Command-Line Tools to the secondary , follow the steps below:

Generate Intel hex files from the output binary (bin) files as follows:

```
srec_cat install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/tfm_s_ns_signed.bin -binary␣
↪--offset=0x10000000 -o install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/tfm_s_ns_
↪signed_qspi.hex -intel
```

- Connect the micro-USB cable to the BL5340 DVK and to your computer
- Erase the flash memory in the BL5340 module:

```
nrfjprog --eraseall -f nrf53
```

- Flash the BL2 and the TF-M image binaries from the sample folder of your choice:

```
nrfjprog --program <sample folder>/install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/
↪bl2.hex -f nrf53 --sectorerase
nrfjprog --program <sample folder>/install/outputs/LAIRDCONNECTIVITY/BL5340_DVK_CPUAPP/
↪tfm_s_ns_signed.hex -f nrf53 --qspisectorerase
```

- Reset and start TF-M:

```
nrfjprog --reset -f nrf53
```

Secure UART Console on BL5340 DVK

SECURE_UART1 is enabled by default when building TF-M on the BL5340 DVK, so the secure firmware console output is available via USART1.

Non-Secure console output is available via USART0.

Note: By default USART0 and USART1 outputs are routed to separate serial ports.

7.5 Nordic platforms

7.5.1 Nordic nRF5340

The nRF5340 development kit (DK) is a single-board development kit for the evaluation and development on the Nordic nRF5340 System-on-Chip (SoC).

The nRF5340 is a dual-core SoC based on the Arm® Cortex®-M33 architecture, with:

- a full-featured ARM Cortex-M33F core with DSP instructions, FPU, and ARMv8-M Security Extension, running at up to 128 MHz, referred to as the **Application MCU**

- a secondary ARM Cortex-M33 core, with a reduced feature set, running at a fixed 64 MHz, referred to as the **Network MCU**.

The nRF5340 Application MCU supports the Armv8m Security Extension.

Documentation

The following links provide useful information about the nRF5340

nRF5340 DK website:
https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF5340-DK

Nordic Semiconductor Infocenter: https://infocenter.nordicsemi.com

Building TF-M on nRF5340 Application MCU

To build an S and NS application image for the nRF5340 Application MCU run the following commands:

```
cmake -S <TF-M base folder> -B build_spe \
        -DTFM_PLATFORM=nordic_nrf/nrf5340dk_nrf5340_cpuapp \
        -DTFM_TOOLCHAIN_FILE=../toolchain_GNUARM.cmake \
cmake --build build_spe -- install

cmake -S <Application base folder> -B build_app \
        -DCONFIG_SPE_PATH<Absolute path to TF-M base folder>/build_spe/api_ns
        -DTFM_TOOLCHAIN_FILE=cmake/toolchain_ns_GNUARM.cmake
cmake --build
```

Note: Currently, applications can only be built using GCC (GNU ARM Embedded toolchain).

Note: BL2 (MCUBoot) logging output to be available, the project needs to be built with Debug configuration (CMAKE_BUILD_TYPE=Debug).

Flashing and debugging with Nordic nRF Segger J-Link

nRF5340 DK is equipped with a Debug IC (Atmel ATSAM3U2C) which provides the following functionality:

- Segger J-Link firmware and desktop tools
- SWD debug for the nRF5340 IC
- Mass Storage device for drag-and-drop image flashing
- USB CDC ACM Serial Port bridged to the nRFx UART peripheral
- Segger RTT Console
- Segger Ozone Debugger

To install the J-Link Software and documentation pack, follow the steps below:

1. Download the appropriate package from the J-Link Software and documentation pack website
2. Depending on your platform, install the package or run the installer

3. When connecting a J-Link-enabled board such as an nRF5340 DK, a drive corresponding to a USB Mass Storage device as well as a serial port should come up

nRF Command-Line Tools Installation

The nRF Command-line Tools allow you to control your nRF5340 device from the command line, including resetting it, erasing or programming the flash memory and more.

To install them, visit nRF Command-Line Tools and select your operating system.

After installing, make sure that `nrfjprog` is somewhere in your executable path to be able to invoke it from anywhere.

BL2, S, and NS application images can be flashed into nRF5340 separately or may be merged together into a single binary.

Flashing the nRF5340 DK

To program the flash with a compiled TF-M image (i.e. S, NS or both) after having followed the instructions to install the Segger J-Link Software and the nRF Command-Line Tools, follow the steps below:

Generate Intel hex files from the output binary (bin) files as follows:

```
srec_cat build_app/tfm_s_ns_signed.bin -binary --offset 0x10000 \
      -o build_app/tfm_s_ns_signed.hex -intel
```

- Connect the micro-USB cable to the nRF5340 DK and to your computer
- Erase the flash memory in the nRF5340 IC:

```
nrfjprog --eraseall -f nrf53
```

- **(Optionally) Erase the flash memory and reset flash protection and disable**
  the read back protection mechanism if enabled.

```
nrfjprog --recover -f nrf53
```

- Flash the BL2 and the TF-M image binaries from the sample folder of your choice:

```
nrfjprog --program build_spe/bin/bl2.hex -f nrf53 --sectorerase
nrfjprog --program build_app/tfm_s_ns_signed.hex -f nrf53 --sectorerase
```

- Reset and start TF-M:

```
nrfjprog --reset -f nrf53
```

Secure UART Console on nRF5340 DK

SECURE_UART1 is enabled by default when building TF-M on nRF5340 DK, so the secure firmware console output is available via USART1. When using the nRF5340 network MCU make sure to configure different UART pins or disable SECURE_UART1.

Non-Secure console output is available via USART0.

Note: By default USART0 and USART1 outputs are routed to separate serial ports.

7.5.2 Nordic nRF9160

The nRF9160 development kit (DK) is a single-board development kit for the evaluation and development on the Nordic nRF9160 SiP for LTE-M and NB-IoT.

The nRF9160 SoC features a full-featured Arm® Cortex®-M33F core with DSP instructions, FPU, and ARMv8-M Security Extension, running at up to 64 MHz.

Documentation

The following links provide useful information about the nRF9160

nRF9160 DK website:
https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF9160-DK

Nordic Semiconductor Infocenter: https://infocenter.nordicsemi.com

Building TF-M on nRF9160

To build an S and NS application image for the nRF9160 run the following commands:

```
cmake -S <TF-M base folder> -B build_spe \
        -DTFM_PLATFORM=nordic_nrf/nrf9160dk_nrf9160 \
        -DTFM_TOOLCHAIN_FILE=../toolchain_GNUARM.cmake \
cmake --build build_spe -- install

cmake -S <Application base folder> -B build_app \
        -DCONFIG_SPE_PATH<Absolute path to TF-M base folder>/build_spe/api_ns
        -DTFM_TOOLCHAIN_FILE=cmake/toolchain_ns_GNUARM.cmake
cmake --build
```

Note: Currently, applications can only be built using GCC (GNU ARM Embedded toolchain).

Note: For BL2 (MCUBoot) logging output to be available, the project needs to be built with Debug configuration (CMAKE_BUILD_TYPE=Debug).

Flashing and debugging with Nordic nRF Segger J-Link

nRF9160 DK is equipped with a Debug IC (Atmel ATSAM3U2C) which provides the following functionality:

- Segger J-Link firmware and desktop tools
- SWD debug for the nRF9160 IC
- Mass Storage device for drag-and-drop image flashing
- USB CDC ACM Serial Port bridged to the nRFx UART peripheral
- Segger RTT Console
- Segger Ozone Debugger

To install the J-Link Software and documentation pack, follow the steps below:

1. Download the appropriate package from the J-Link Software and documentation pack website
2. Depending on your platform, install the package or run the installer
3. When connecting a J-Link-enabled board such as an nRF9160 DK, a drive corresponding to a USB Mass Storage device as well as a serial port should come up

nRFx Command-Line Tools Installation

The nRF Command-line Tools allow you to control your nRF9160 device from the command line, including resetting it, erasing or programming the flash memory and more.

To install them, visit nRF Command-Line Tools and select your operating system.

After installing, make sure that `nrfjprog` is somewhere in your executable path to be able to invoke it from anywhere.

BL2, S, and NS application images can be flashed into nRF9160 separately or may be merged together into a single binary.

Flashing the nRF9160 DK

To program the flash with a compiled TF-M image (i.e. S, NS or both) after having followed the instructions to install the Segger J-Link Software and the nRFx Command-Line Tools, follow the steps below:

Generate Intel hex files from the output binary (bin) files as follows:

```
srec_cat build_app/tfm_s_ns_signed.bin -binary --offset 0x10000 \
      -o build_app/tfm_s_ns_signed.hex -intel
```

- Connect the micro-USB cable to the nRF9160 DK and to your computer
- Erase the flash memory in the nRF9160 IC:

```
nrfjprog --eraseall -f nrf91
```

- **(Optionally) Erase the flash memory and reset flash protection and disable**
  the read back protection mechanism if enabled.

```
nrfjprog --recover -f nrf91
```

- Flash the BL2 and TF-M image binaries from the sample folder of your choice:

```
nrfjprog --program build_spe/bin/bl2.hex -f nrf91 --sectorerase
nrfjprog --program build_app/tfm_s_ns_signed.hex -f nrf91 --sectorerase
```

- Reset and start TF-M:

```
nrfjprog --reset -f nrf91
```

Secure UART Console on nRF9160 DK

SECURE_UART1 is enabled by default when building TF-M on nRF9160 DK, so the secure firmware console output is available via USART1.

Non-Secure console output is available via USART0.

Note: By default USART0 and USART1 outputs are routed to separate serial ports.

7.5.3 Nordic nRF9161

The nRF9161 development kit (DK) is a single-board development kit for the evaluation and development on the Nordic nRF9161 SiP for LTE-M and NB-IoT.

The nRF9161 SoC features a full-featured Arm® Cortex®-M33F core with DSP instructions, FPU, and ARMv8-M Security Extension, running at up to 64 MHz.

Documentation

The following links provide useful information about the nRF9161

nRF9161 DK website:
https://www.nordicsemi.com/products/nrf9161

Nordic Semiconductor Infocenter: https://infocenter.nordicsemi.com

Building TF-M on nRF9161

To build an S and NS application image for the nRF9161 run the following commands:

```
cmake -S <TF-M base folder> -B build_spe \
        -DTFM_PLATFORM=nordic_nrf/nrf9161dk_nrf9161 \
        -DTFM_TOOLCHAIN_FILE=../toolchain_GNUARM.cmake \
cmake --build build_spe -- install

cmake -S <Application base folder> -B build_app \
        -DCONFIG_SPE_PATH<Absolute path to TF-M base folder>/build_spe/api_ns
        -DTFM_TOOLCHAIN_FILE=cmake/toolchain_ns_GNUARM.cmake
cmake --build
```

Note: Currently, applications can only be built using GCC (GNU ARM Embedded toolchain).

Note: For BL2 (MCUBoot) logging output to be available, the project needs to be built with Debug configuration (CMAKE_BUILD_TYPE=Debug).

Flashing and debugging with Nordic nRF Segger J-Link

nRF9161 DK is equipped with a Debug IC (Atmel ATSAM3U2C) which provides the following functionality:

- Segger J-Link firmware and desktop tools
- SWD debug for the nRF9161 IC
- Mass Storage device for drag-and-drop image flashing
- USB CDC ACM Serial Port bridged to the nRFx UART peripheral
- Segger RTT Console
- Segger Ozone Debugger

To install the J-Link Software and documentation pack, follow the steps below:

1. Download the appropriate package from the J-Link Software and documentation pack website
2. Depending on your platform, install the package or run the installer
3. When connecting a J-Link-enabled board such as an nRF9161 DK, a drive corresponding to a USB Mass Storage device as well as a serial port should come up

nRFx Command-Line Tools Installation

The nRF Command-line Tools allow you to control your nRF9161 device from the command line, including resetting it, erasing or programming the flash memory and more.

To install them, visit nRF Command-Line Tools and select your operating system.

After installing, make sure that `nrfjprog` is somewhere in your executable path to be able to invoke it from anywhere.

BL2, S, and NS application images can be flashed into nRF9161 separately or may be merged together into a single binary.

Flashing the nRF9161 DK

To program the flash with a compiled TF-M image (i.e. S, NS or both) after having followed the instructions to install the Segger J-Link Software and the nRFx Command-Line Tools, follow the steps below:

Generate Intel hex files from the output binary (bin) files as follows:

```
srec_cat build_app/tfm_s_ns_signed.bin -binary --offset 0x10000 \
      -o build_app/tfm_s_ns_signed.hex -intel
```

- Connect the micro-USB cable to the nRF9161 DK and to your computer
- Erase the flash memory in the nRF9161 IC:

```
nrfjprog --eraseall -f nrf91
```

- **(Optionally) Erase the flash memory and reset flash protection and disable**
  the read back protection mechanism if enabled.

```
nrfjprog --recover -f nrf91
```

- Flash the BL2 and TF-M image binaries from the sample folder of your choice:

```
nrfjprog --program build_spe/bin/bl2.hex -f nrf91 --sectorerase
nrfjprog --program build_app/tfm_s_ns_signed.hex -f nrf91 --sectorerase
```

- Reset and start TF-M:

```
nrfjprog --reset -f nrf91
```

Secure UART Console on nRF9161 DK

SECURE_UART1 is enabled by default when building TF-M on nRF9161 DK, so the secure firmware console output is available via USART1.

Non-Secure console output is available via USART0.

Note: By default USART0 and USART1 outputs are routed to separate serial ports.

7.6 Nuvoton platforms

7.6.1 NuMaker-PFM-M2351

Building TF-M

Build TF-M regression test with M2351 by following commands:

```
$ cd tf-m-test/tests_reg
$ cmake -S spe -B build_spe \
        -G"Unix Makefiles" \
        -DTFM_PLATFORM=nuvoton/m2351 \
        -DTFM_TOOLCHAIN_FILE=[tf-m path]/toolchain_GNUARM.cmake \
        -DCMAKE_BUILD_TYPE=Release \
        -DTEST_S=ON -DTEST_NS=ON
$ cmake --build build --parallel -- install
$ cmake -S . -B build_test \
        -G"Unix Makefiles" \
        -DCONFIG_SPE_PATH=[tf-m-tests path]/tests_reg/build_spe/api_ns \
        -DTFM_TOOLCHAIN_FILE=cmake/toolchain_ns_GNUARM.cmake \
        -DCMAKE_BUILD_TYPE=Release \
$ cmake --build build_test --parallel -- install
```

Define TEST_NS=ON or TEST_S=ON for non-secure or secure regression test. CMAKE_BUILD_TYPE could be “Release”, “RelWithDebInfo” or “Minsizerel” Other cmake parameters should not be changed.

Flashing Image with Nuvoton NuLink Tool

NuMaker-PFM-M2351 board supports on board ICE called NuLink. Connect NuLink to PC with USB cable and using NuLink Command Tool to flashing the image.

The commands are as follows:

```
> NuLink_M2351_M261 -C
> NuLink_M2351_M261 -E ALL
> NuLink_M2351_M261 -W NSCBA 0x80000 0
> NuLink_M2351_M261 -S
> NuLink_M2351_M261 -C
> NuLink_M2351_M261 -W APROM .\build_spe\bin\bl2.bin 0
> NuLink_M2351_M261 -W APROM .\build_spe\bin\tfm_s_signed.bin 0 0x20000 0
> NuLink_M2351_M261 -W APROMNS .\build_test\bin\tfm_ns_signed.bin 0
```

7.6.2 NuMaker-PFM-M2354

Building TF-M

Build TF-M regression test with M2354 by following commands:

```
$ cd tf-m-test/tests_reg
$ cmake -S spe -B build_spe \
        -G"Unix Makefiles" \
        -DTFM_PLATFORM=nuvoton/m2354 \
        -DTFM_TOOLCHAIN_FILE=[tf-m path]/toolchain_GNUARM.cmake \
        -DCMAKE_BUILD_TYPE=Release \
        -DTEST_S=ON -DTEST_NS=ON
$ cmake --build build --parallel -- install
$ cmake -S . -B build_test \
        -G"Unix Makefiles" \
        -DCONFIG_SPE_PATH=[tf-m-tests path]/tests_reg/build_spe/api_ns \
        -DTFM_TOOLCHAIN_FILE=cmake/toolchain_ns_GNUARM.cmake \
        -DCMAKE_BUILD_TYPE=Release \
$ cmake --build build_test --parallel -- install
```

Define TEST_NS=ON or TEST_S=ON for non-secure or secure regression test. CMAKE_BUILD_TYPE could be “Release”, “RelWithDebInfo” or “Minsizerel” Other cmake parameters should not be changed.

Flashing Image with Nuvoton NuLink Tool

NuMaker-PFM-M2354 board supports on board ICE called NuLink. Connect NuLink to PC with USB cable and using NuLink Command Tool to flashing the image.

The commands are as follows:

```
> NuLink_M2354 -C
> NuLink_M2354 -E ALL
> NuLink_M2354 -W NSCBA 0x80000 0
> NuLink_M2354 -S
> NuLink_M2354 -C
> NuLink_M2354 -W APROM .\build_spe\bin\bl2.bin 0
> NuLink_M2354 -W APROM .\build_spe\bin\tfm_s_signed.bin 0 0x20000 0
> NuLink_M2354 -W APROMNS .\build_test\bin\tfm_ns_signed.bin 0
```

7.7 NXP platform

7.7.1 LPCXpresso55S69

1. Building TF-M

Prepare the tf-m-tests repository inside the TF-M base folder.

```
cd <TF-M base folder>
git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
```

There are two options for the TF-M build - with or without secondary bootloader (BL2).

1.1 Building TF-M demo without BL2

To build S and NS application image without a BL2, run the following commands:

```
cd <TF-M base folder>/tf-m-test/tests_reg
cmake -S spe -B build_spe -DTFM_PLATFORM=nxp/lpcxpresso55s69 -DCONFIG_TFM_SOURCE_PATH=
↪<TF-M base folder>/trusted-firmware-m -G"Unix Makefiles" -DBL2=OFF
cmake --build build_spe -- install
cmake -S . -B build_test -G"Unix Makefiles" -DCONFIG_SPE_PATH=<TF-M base folder>/tf-m-
↪tests/tests_reg/build_spe/api_ns
cmake --build build_test
```

1.2 Building TF-M demo with BL2

To build S and NS application along with a BL2, run the following commands:

```
cd <TF-M base folder>/tf-m-test/tests_reg
cmake -S spe -B build_spe -DTFM_PLATFORM=nxp/lpcxpresso55s69 -DCONFIG_TFM_SOURCE_PATH=
↪<TF-M base folder>/trusted-firmware-m -G"Unix Makefiles" -DBL2=ON
cmake --build build_spe -- install
cmake -S . -B build_test -G"Unix Makefiles" -DCONFIG_SPE_PATH=<TF-M base folder>/tf-m-
↪tests/tests_reg/build_spe/api_ns
cmake --build build_test
```

1.3 Building TF-M regression tests

To buld the S and NS regression tests without BL2, run the following commands:

```
cd <TF-M base folder>/tf-m-test/tests_reg
cmake -S spe -B build_spe -DTFM_PLATFORM=nxp/lpcxpresso55s69 -DCONFIG_TFM_SOURCE_PATH=
↪<TF-M base folder>/trusted-firmware-m -G"Unix Makefiles" -DTEST_S=ON -DTEST_NS=ON -
↪DBL2=OFF
cmake --build build_spe -- install
cmake -S . -B build_test -G"Unix Makefiles" -DCONFIG_SPE_PATH=<TF-M base folder>/tf-m-
↪tests/tests_reg/build_spe/api_ns
cmake --build build_test
```

2. Flashing TF-M with PyOCD

PyOCD is an open source Python package for programming and debugging Arm Cortex-M microcontrollers using multiple supported types of USB debug probes. See: *PyOCD <https://pypi.org/project/pyocd/>*

If you built TF-M with the BL2 secondary bootloader, use the following commands:

```
pyocd erase --mass -t LPC55S69
pyocd flash build_spe/bin/bl2.hex -t LPC55S69
pyocd flash build_test/tfm_s_ns_signed.bin --base-address 0x8000 -t LPC55S69
```

When BL2 is disabled, flash the generated hex secure and non-secure images:

```
pyocd erase --mass -t LPC55S69
pyocd flash build_spe/bin/tfm_s.hex build_test/bin/tfm_ns.hex -t LPC55S69
```

Note: There is sometimes a stability issue with the pyocd flash erasing, it is needed to rerun the pyocd erase command again.

7.8 STMICROELECTRONICS

This document contains SOC and boards supported by stm32.

7.8.1 STM32H5

TF-M is supported on STM32H5 family

https://www.st.com/en/microcontrollers-microprocessors/stm32h5-series.html

Directory content

- **stm/common/stm32h5xx/stm32h5xx_hal:**
  Content from https://github.com/STMicroelectronics/stm32h5xx_hal_driver
- **stm/common/stm32h5xx/Device:**
  Content from https://github.com/STMicroelectronics/cmsis_device_h5
- **stm/common/stm32h5xx/bl2:**
  stm32h5xx bl2 code specific from https://github.com/STMicroelectronics/STM32CubeH5.git (Projects/STM32H573I_DK/Applications/TFM)
- **stm/common/stm32h5xx/secure:**
  stm32h5xx Secure porting adaptation from https://github.com/STMicroelectronics/STM32CubeH5.git (Projects/STM32H573I_DK/Applications/TFM)
- **stm/common/stm32h5xx/boards:**
  Adaptation and tools specific to stm32 board using stm32h5xx soc from https://github.com/STMicroelectronics/STM32CubeH5.git (Projects/STM32H573I_DK/Applications/TFM)
- **stm/common/stm32h5xx/CMSIS_Driver:**
  Flash and uart driver for stm32h5xx platform
- **stm/common/stm32h5xx/Native_Driver:**
  Random generator and tickless implementation

Specific Software Requirements

STM32_Programmer_CLI is required.(see https://www.st.com/en/development-tools/stm32cubeprog.html)

Limitations to Consider When Using the Platform

MPU and ICACHE disabled in bl2 boot stage

Write software on target

In build folder:

- `postbuild.sh`: Updates regression.sh and TFM_UPDATE.sh scripts according to flash_layout.h
- `regression.sh`: Sets platform option bytes config and erase platform
- `TFM_UPDATE.sh`: Writes bl2, secure, and non secure image in target

Connect board to USB and Execute the 3 scripts in following order to update platform: postbuild.sh, regression.sh, TFM_UPDATE.sh

The virtual com port from STLINK is used for TFM log and serial port configuration should be:

- Baud rate = 115200
- Data = 8 bits
- Parity = none
- Stop = 1 bit
- Flow control = none

7.8.2 STM32L5

TF-M is supported on STM32L5 family

https://www.st.com/en/microcontrollers-microprocessors/stm32l5-series.html

https://www.st.com/resource/en/user_manual/dm00678763-getting-started-with-stm32cubel5-tfm-application-stmicroelectronics.pdf

Directory content

- **stm/common/stm32l5xx/stm32l5xx_hal:**
  Content from https://github.com/STMicroelectronics/stm32l5xx_hal_driver.git
- **stm/common/stm32l5xx/Device:**
  Content from https://github.com/STMicroelectronics/cmsis_device_l5.git
- **stm/common/stm32l5xx/bl2:**
  stm32l5xx bl2 code specific from https://github.com/STMicroelectronics/STM32CubeL5.git (Projects/STM32L562E-DK/Applications/TFM)
- **stm/common/stm32l5xx/secure:**
  stm32l5xx Secure porting adaptation from https://github.com/STMicroelectronics/STM32CubeL5.git (Projects/STM32L562E-DK/Applications/TFM)
- **stm/common/stm32l5xx/boards:**
  Adaptation and tools specific to stm32 board using stm32l5xx soc from https://github.com/STMicroelectronics/STM32CubeL5.git (Projects/STM32L562E-DK/Applications/TFM)
- **stm/common/stm32l5xx/CMSIS_Driver:**
  Flash and uart driver for stm32l5xx platform
- **stm/common/stm32l5xx/Native_Driver:**
  Random generator and tickless implementation

Specific Software Requirements

STM32_Programmer_CLI is required.(see https://www.st.com/en/development-tools/stm32cubeprog.html)

Write software on target

In secure build folder directory api_ns:

- `postbuild.sh`: Updates regression.sh and TFM_UPDATE.sh scripts according to flash_layout.h
- `regression.sh`: Sets platform option bytes config and erase platform
- `TFM_UPDATE.sh`: Writes bl2, secure, and non secure image in target

Connect board to USB and Execute the 3 scripts in following order to update platform: postbuild.sh, regression.sh, TFM_UPDATE.sh

The virtual com port from STLINK is used for TFM log and serial port configuration should be:

- Baud rate = 115200
- Data = 8 bits
- Parity = none
- Stop = 1 bit
- Flow control = none

7.8.3 STM32U5

TF-M is supported on STM32U5 family

https://www.st.com/en/microcontrollers-microprocessors/stm32u5-series.html

Directory content

- **stm/common/stm32u5xx/stm32u5xx_hal:**
  Content from https://github.com/STMicroelectronics/stm32u5xx_hal_driver.git
- **stm/common/stm32u5xx/Device:**
  Content from https://github.com/STMicroelectronics/cmsis_device_u5.git
- **stm/common/stm32u5xx/bl2:**
  stm32l5xx bl2 code specific from https://github.com/STMicroelectronics/STM32CubeU5.git (Projects/B-U585I-IOT02A/Applications/TFM)
- **stm/common/stm32u5xx/secure:**
  stm32l5xx Secure porting adaptation from https://github.com/STMicroelectronics/STM32CubeU5.git (Projects/B-U585I-IOT02A/Applications/TFM)
- **stm/common/stm32u5xx/boards:**
  Adaptation for stm32 board using stm32l5xx soc from https://github.com/STMicroelectronics/STM32CubeU5.git (Projects/B-U585I-IOT02A/Applications/TFM)

Specific Software Requirements

STM32_Programmer_CLI is required.(see https://www.st.com/en/development-tools/stm32cubeprog.html)

Write software on target

In secure build folder directory api_ns:

- `postbuild.sh`: Updates regression.sh and TFM_UPDATE.sh scripts according to flash_layout.h
- `regression.sh`: Sets platform option bytes config and erase platform
- `TFM_UPDATE.sh`: Writes bl2, secure, and non secure image in target

Connect board to USB and Execute the 3 scripts in following order to update platform: postbuild.sh, regression.sh, TFM_UPDATE.sh

The virtual com port from STLINK is used for TFM log and serial port configuration should be:

- Baud rate = 115200
- Data = 8 bits
- Parity = none
- Stop = 1 bit
- Flow control = none

7.8.4 NUCLEO_L552ZE_Q

STM32 Nucleo-144 development board with STM32L552ZE MCU, SMPS, supports Arduino, ST Zio and morpho connectivity https://www.st.com/en/evaluation-tools/nucleo-l552ze-q.html

Configuration and Build

GNUARM/ARMCLANG/IARARM compilation is available for this target. and build the selected configuration as follow.

The build configuration for TF-M is provided to the build system using command line arguments. Required arguments are noted below.

The following instructions build multi-core TF-M with regression test suites in Isolation Level 1.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_reg/spe -B . -GNinja -DTFM_PLATFORM=stm/nucleo_l552ze_q
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
```

(continues on next page)

(continued from previous page)

```
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_S=ON -DTEST_NS=ON

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_reg -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

The following instructions build multi-core TF-M with PSA API test suite for the attestation service in Isolation Level 1 on Linux.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_psa_arch/spe -B . -GNinja -DTFM_PLATFORM=stm/nucleo_l552ze_
↪q
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_PSA_API=INITIAL_ATTESTATION

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_psa_arch -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

7.8.5 STM32H573I_DK

Discovery kit for IoT node with STM32H5 series https://www.st.com/en/evaluation-tools/stm32h573i-dk.html

Configuration and Build

GNUARM/ARMCLANG/IARARM compilation is available for this target. and build the selected configuration as follow.

The build configuration for TF-M is provided to the build system using command line arguments. Required arguments are noted below.

The following instructions build multi-core TF-M with regression test suites in Isolation Level 1.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_reg/spe -B . -GNinja -DTFM_PLATFORM=stm/stm32h573i_dk
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_S=ON -DTEST_NS=ON

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_reg -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

The following instructions build multi-core TF-M with PSA API test suite for the attestation service in Isolation Level 1 on Linux.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_psa_arch/spe -B . -GNinja -DTFM_PLATFORM=stm/stm32h573i_dk
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_PSA_API=INITIAL_ATTESTATION
```

(continues on next page)

(continued from previous page)

```
ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_psa_arch -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

7.8.6 STM32L562E_DK

Discovery kit with STM32L562QE MCU : https://www.st.com/en/evaluation-tools/stm32l562e-dk.html

Configuration and Build

GNUARM/ARMCLANG/IARARM compilation is available for this target. and build the selected configuration as follow.

The build configuration for TF-M is provided to the build system using command line arguments. Required arguments are noted below.

The following instructions build multi-core TF-M with regression test suites in Isolation Level 1.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_reg/spe -B . -GNinja -DTFM_PLATFORM=stm/stm32l562e_dk
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_S=ON -DTEST_NS=ON

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_reg -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

The following instructions build multi-core TF-M with PSA API test suite for the attestation service in Isolation Level 1 on Linux.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_psa_arch/spe -B . -GNinja -DTFM_PLATFORM=stm/stm32l562e_dk
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_PSA_API=INITIAL_ATTESTATION

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_psa_arch -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

7.8.7 B_U585I_IOT02A

Discovery kit for IoT node with STM32U5 series https://www.st.com/en/evaluation-tools/B-U585I-IOT02A.html

Configuration and Build

GNUARM/ARMCLANG/IARARM compilation is available for this target. and build the selected configuration as follow.

The build configuration for TF-M is provided to the build system using command line arguments. Required arguments are noted below.

The following instructions build multi-core TF-M with regression test suites in Isolation Level 1.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

```

(continues on next page)

(continued from previous page)

```
cmake -S /../tf-m-tests/tests_reg/spe -B . -GNinja -DTFM_PLATFORM=stm/b_u585i_iot02a
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_S=ON -DTEST_NS=ON

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_reg -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

The following instructions build multi-core TF-M with PSA API test suite for the attestation service in Isolation Level 1 on Linux.

```
cd <TF-M base folder>
cd <trusted-firmware-m folder>

git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>

mkdir build_s && cd build_s

cmake -S /../tf-m-tests/tests_psa_arch/spe -B . -GNinja -DTFM_PLATFORM=stm/b_u585i_iot02a
     -DTFM_TOOLCHAIN_FILE= /../toolchain_ARMCLANG.cmake
     -DCONFIG_TFM_SOURCE_PATH= /../trusted-firmware-m
     -DTFM_PSA_API=ON -DTFM_ISOLATION_LEVEL=1
     -DTEST_PSA_API=INITIAL_ATTESTATION

ninja -C . install -j 8

cd <trusted-firmware-m folder>
mkdir build_ns && cd build_ns
cmake -S /../trusted-firmware-m  /../tf-m-tests/tests_psa_arch -B . -GNinja
     -DCONFIG_SPE_PATH= /../build_s/api_ns -DTFM_TOOLCHAIN_FILE= /../build_s/api_ns/
↪cmake/toolchain_ns_ARMCLANG.cmake

ninja -C . -j 8
```

CHAPTER

EIGHT

BUILD INSTRUCTIONS

Warning: The build process was changed a lot in Q3 2023 and included into the release v2.0. For building instructions for early versions please refer to the documentation of respective versions.

As you know from the introduction TF-M implements *SPE* with a set of secure services. TF-M application as *NSPE* client uses those services through isolation boundary via *PSA-FF-M* API. Both SPE and NSPE are separate binaries and built independently. SPE and NSPE binaries are combined and signed making the final image for downloading onto targets when building NSPE.

Note: This document describes the process of building a single SPE alone. Refer to *Building Tests* on how to build TF-M regression tests and PSA Arch tests to verify TF-M.

TF-M uses CMake **v3.15** or higher. Before starting please make sure you have all required software installed and configured as explained in the TF-M getting started.

Contents

The additional building materials you can find in the following links:TF-M source folder

8.1 Build Tests

The tests is a TF-M application which verifies TF-M functionality on both SPE and NSPE sides. Thus, tests require an extension of SPE side with test code and extra functionality for some Non-Secure test cases. To inject that test code into SPE the `CONFIG_TFM_TEST_DIR` option is used. When SPE build system sees this option it adds the corresponding folder via `add_subdirectory(${CONFIG_TFM_TEST_DIR} tf-m-tests)` and includes it to SPE binary. Also, test configurations should be passed to SPE build to include building Secure Tests.

To hide these complexities to developers, TF-M implements a wrapper CMake in **tf-m-tests** repository[1] to build the SPE for testing rather than building it from the TF-M repository.

The recommended tf-m-tests repo commit to verify TF-M can be found at `<TF-M source dir>/lib/ext/tf-m-tests/version.txt`. It does not support auto-downloading as builds start from it. You need to download it manually before building any tests with the following commands:

```
git clone https://git.trustedfirmware.org/TF-M/tf-m-tests.git
git checkout <recommended tf-m-tests commit>
```

8.1.1 Regression Tests

For instructions on configuring, building and executing the regression tests please refer to the documentation in **tf-m-tests** repository (to be added). The regression test application is located under **/tests_reg** folder. It is recommended to build both SPE and NSPE from that folder.

The basic commands for building the regression tests will be:

```
cd </tf-m-tests/tests_reg>
cmake -S spe -B build_spe -DTFM_PLATFORM=arm/mps2/an521 \
      -DCONFIG_TFM_SOURCE_PATH=<TF-M source dir> \
      -DTFM_TOOLCHAIN_FILE=<Absolute path to>/toolchain_ARMCLANG.cmake \
      -DTEST_S=ON -DTEST_NS=ON
cmake --build build_spe -- install

cmake -S . -B build_test -DCONFIG_SPE_PATH=<Absolute path to>/build_spe/api_ns
cmake --build build_test
```

Instead of enable all the supported Secure (`TEST_S`) and NS (`TEST_NS`) tests, you can also enable individual test suites by using `-DTEST_S_<SUITE>=ON` or `-DTEST_NS_<SUITE>=ON`. For the available test suites, refer to the `default_s_test_config.cmake` and `default_ns_test_config.cmake` files in tf-m-tests repo.

Note: All the test suite config options should be passed to the SPE build command, including NS ones. The SPE building command also accepts all the other config options used to build a single TF-M. All options passing to SPE build does not have to be duplicated in NSPE build, including NS test suite config options. And this also applies to the below *PSA API tests*.

[1] tf-m-tests

8.1.2 PSA API tests

PSA API tests from https://github.com/ARM-software/psa-arch-tests use the same mechanism for SPE extension as the regression tests above utilising `CONFIG_TFM_TEST_DIR` option. PSA API tests are selected by the TEST_PSA_API variable. Enabling both regression tests and PSA API tests simultaneously is **not** supported.

TF-M implements a wrapper CMake for PSA API tests as well. The PSA API test codes are located under **/tests_psa_arch** folder.

Here is a brief description of the basic flow: Select one of the following test suites to be run.

```
-DTEST_PSA_API=INTERNAL_TRUSTED_STORAGE
-DTEST_PSA_API=PROTECTED_STORAGE
-DTEST_PSA_API=STORAGE
-DTEST_PSA_API=CRYPTO
-DTEST_PSA_API=INITIAL_ATTESTATION
-DTEST_PSA_API=IPC
```

Respectively for the corresponding service. For detailed information, please refer to **PSA Certified APIs Architecture Test Suite** section[2] in the documentation of **psa-arch-tests** repository.

For example, to enable the PSA API tests for the Crypto service:

```
cd </tf-m-tests/tests_psa_arch folder>
cmake -S spe -B build_spe -DTFM_PLATFORM=arm/mps2/an521 -DCONFIG_TFM_SOURCE_PATH=<TF-M␣
↪source dir>
      -DTEST_PSA_API=CRYPTO
cmake --build build_spe -- install

cmake -S . -B build_test -DCONFIG_SPE_PATH=<Absolute path to>/build_spe/api_ns
cmake --build build_test
```

Reference

8.2 Run TF-M tests and applications on Arm platforms

Instructions for how to run TF-M tests and applications on Arm platforms.

Follow *build instruction* to build the binaries. Follow *secure boot* to build the binaries with or without BL2 bootloader.

[2] PSA Certified APIs Architecture Test Suite

8.2.1 Execute TF-M regression tests on MPS2 boards and FVP

The BL2 bootloader and TF-M example application and tests have been verified using the reference model for MPS2 (AN521), in Keil MDK , Fixed Virtual Platforms and Arm Development Studio .

Note: The name of the reference model's executable can vary depending on toolchain.

- SMM-SSE-200 for Keil MDK
- FVP_MPS2_AEMv8M for Fixed Virtual Platforms and Arm Development Studio

For more information please refer to the appropriate toolchain's documentation: Keil MDK Documentation , Fixed Virtual Platforms Documentation , Arm Development Studio Documentation

To run the example code on an SSE-200 Fast-Model

Using FVP_MPS2_AEMv8M provided by Arm Development Studio 2019.1.

Example application and regression tests without BL2 bootloader

Add `tfm_s.axf` and `tfm_ns.axf` to symbol files in Debug Configuration menu.

```
<DS_PATH>/sw/models/bin/FVP_MPS2_AEMv8M \
--parameter fvp_mps2.platform_type=2 \
--parameter cpu0.baseline=0 \
--parameter cpu0.INITVTOR_S=0x10000000 \
--parameter cpu0.semihosting-enable=0 \
--parameter fvp_mps2.DISABLE_GATING=0 \
--parameter fvp_mps2.telnetterminal0.start_telnet=1 \
--parameter fvp_mps2.telnetterminal1.start_telnet=0 \
--parameter fvp_mps2.telnetterminal2.start_telnet=0 \
--parameter fvp_mps2.telnetterminal0.quiet=0 \
--parameter fvp_mps2.telnetterminal1.quiet=1 \
--parameter fvp_mps2.telnetterminal2.quiet=1 \
--application cpu0=<build_dir>/bin/tfm_s.axf \
--data cpu0=<build_dir>/bin/tfm_ns.bin@0x00100000
```

Example application and regression tests with BL2 bootloader

To test TF-M with bootloader, one must apply the following changes:

- Add `bl2.axf` to symbol files in DS-5 in Debug Configuration menu.
- Replace the last two lines of the previous command with this:

```
--application cpu0=<build_dir>/bin/bl2.axf \
--data cpu0=<build_dir>/tfm_s_ns_signed.bin@0x10080000
```

Test software upgrade with BL2 bootloader

BL2 bootloader is mandatory to test software update. Furthermore, two TF-M blobs must be built. Outputs of example application and regression test can be used to test it. Load output of example application to the primary slot (0x10080000) and output of regression test to the secondary slot (0x10180000). Add the following line to the end of the previous chapter:

```
--data cpu0=<build_dir>/tfm_s_ns_signed.bin@0x10180000
```

To run the example code on SSE 200 FPGA on MPS2 board

FPGA image is available to download here

To run BL2 bootloader and TF-M example application and tests in the MPS2 board, it is required to have SMM-SSE-200 for MPS2 (AN521) image in the MPS2 board SD card. The image should be located in `<MPS2 device name>/MB/HBI0263<board revision letter>/AN521`

The MPS2 board tested is HBI0263C referred also as MPS2+.

Warning: If you change the exe names, MPS2 expects file names in 8.3 format.

Example application

1. Copy `bl2.bin` and `tfm_s_ns_signed.bin` files from `<build_dir>/bin` to `<MPS2 device name>/SOFTWARE/`
2. Open `<MPS2 device name>/MB/HBI0263C/AN521/images.txt`
3. Update the `AN521/images.txt` file as follows:

```
TITLE: Versatile Express Images Configuration File
[IMAGES]
TOTALIMAGES: 2                     ;Number of Images (Max: 32)
IMAGE0ADDRESS: 0x10000000
IMAGE0FILE: \Software\bl2.bin  ; BL2 bootloader
IMAGE1ADDRESS: 0x10080000
IMAGE1FILE: \Software\tfm_s_ns_signed.bin ; TF-M example application binary blob
```

4. Close `<MPS2 device name>/MB/HBI0263C/AN521/images.txt`
5. Unmount/eject the `<MPS2 device name>` unit
6. Reset the board to execute the TF-M example application
7. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Image 0: magic=good, copy_done=0xff, image_ok=0xff
[INF] Scratch: magic=bad, copy_done=0x5, image_ok=0xcf
[INF] Boot source: primary slot
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x80000
```

(continues on next page)

(continued from previous page)

```
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!
```

Regression tests

After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Image 0: magic=good, copy_done=0xff, image_ok=0xff
[INF] Scratch: magic=bad, copy_done=0x5, image_ok=0xcf
[INF] Boot source: primary slot
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x80000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!

#### Execute test suites for the protected storage service ####
Running Test Suite PS secure interface tests (TFM_PS_TEST_2XXX)...

> Executing 'TFM_PS_TEST_2001'
  Description: 'Create interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2002'
  Description: 'Get handle interface (DEPRECATED)'
This test is DEPRECATED and the test execution was SKIPPED
  TEST PASSED!
> Executing 'TFM_PS_TEST_2003'
  Description: 'Get handle with null handle pointer (DEPRECATED)'
This test is DEPRECATED and the test execution was SKIPPED
  TEST PASSED!
> Executing 'TFM_PS_TEST_2004'
  Description: 'Write interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2005'
  Description: 'Read interface'
....
```

Note: PS reliability tests take a few minutes to run on the MPS2.

Example application without BL2 bootloader

1. Copy `tfm_s.bin` and `tfm_ns.bin` files from `<build_dir>/bin` to `<MPS2 device name>/SOFTWARE/`
2. Open `<MPS2 device name>/MB/HBI0263C/AN521/images.txt`
3. Update the `AN521/images.txt` file as follows:

```
TITLE: Versatile Express Images Configuration File
[IMAGES]
TOTALIMAGES: 2                     ;Number of Images (Max: 32)
IMAGE0ADDRESS: 0x10000000
IMAGE0FILE: \Software\tfm_s.bin  ; Secure code
IMAGE1ADDRESS: 0x00100000
IMAGE1FILE: \Software\tfm_ns.bin ; Non-secure code
```

4. Close `<MPS2 device name>/MB/HBI0263C/AN521/images.txt`
5. Unmount/eject the `<MPS2 device name>` unit
6. Reset the board to execute the TF-M example application
7. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[Sec Thread] Secure image initializing!
```

Regression tests without BL2 bootloader

After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[Sec Thread] Secure image initializing!

#### Execute test suites for the protected storage service ####
Running Test Suite PS secure interface tests (TFM_PS_TEST_2XXX)...

> Executing 'TFM_PS_TEST_2001'
  Description: 'Create interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2002'
  Description: 'Get handle interface (DEPRECATED)'
This test is DEPRECATED and the test execution was SKIPPED
  TEST PASSED!
> Executing 'TFM_PS_TEST_2003'
  Description: 'Get handle with null handle pointer (DEPRECATED)'
This test is DEPRECATED and the test execution was SKIPPED
  TEST PASSED!
> Executing 'TFM_PS_TEST_2004'
  Description: 'Write interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2005'
  Description: 'Read interface'
....
```

8.2.2 Execute TF-M regression tests on Musca test chip boards

Note: Before executing any images on Musca-B1 board, please check the *target platform readme* to have the correct setup.

Install SRecord for Musca test chip boards:

- Windows: SRecord v1.63
- Linux: `sudo apt-get install -y srecord`

Example application with BL2 bootloader

1. Create a unified hex file comprising of both `bl2.bin` and `tfm_s_ns_signed.bin` for Musca-B1 or Musca-S1 platforms:

```
srec_cat bin/bl2.bin -Binary -offset 0xA000000 bin/tfm_s_ns_signed.bin -Binary -
↪offset 0xA020000 -o tfm.hex -Intel
```

2. Power up the Musca board by connecting it to a computer with a USB lead. Press the `PBON` button if the green `ON` LED does not immediately turn on. The board should appear as a USB drive.
3. Copy `tfm.hex` to the USB drive. The orange `PWR` LED should start blinking.
4. Once the `PWR` LED stops blinking, power cycle or reset the board to boot from the new image.
5. After completing the procedure you should see the following messages on the DAPLink UART (baud 115200 8n1):

```
[INF] Starting bootloader
[INF] Image 0: magic=good, copy_done=0xff, image_ok=0xff
[INF] Scratch: magic=bad, copy_done=0x5, image_ok=0xd9
[INF] Boot source: primary slot
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x20000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!
```

Regression tests with BL2 bootloader

Note: As the Internal Trusted Storage and Protected Storage tests use persistent storage, it is recommended to erase the storage area before running the tests. Existing data may prevent the tests from running to completion if, for example, there is not enough free space for the test data or the UIDs used by the tests have already been created with the write-once flag set. Repeated test runs can be done without erasing between runs.

To erase the storage when flashing an image, `-fill 0xFF <start_addr> <end_addr>` can be added to the `srec_cat` command used to create the combined hex file. The `<start_addr>` and `<end_addr>` are the start and end addresses of the storage area, found in the board's `flash_layout.h` file. The board's flash can also be erased via a debugger; see your IDE's documentation for instructions.

After completing the procedure you should see the following messages on the DAPLink UART (baud 115200 8n1):

```
[INF] Starting bootloader
[INF] Image 0: magic=good, copy_done=0xff, image_ok=0xff
[INF] Scratch: magic=bad, copy_done=0x5, image_ok=0x9
[INF] Boot source: primary slot
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x20000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!

#### Execute test suites for the protected storage service ####
Running Test Suite PS secure interface tests (TFM_PS_TEST_2XXX)...
> Executing 'TFM_PS_TEST_2001'
  Description: 'Create interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2002'
  Description: 'Get handle interface (DEPRECATED)'
This test is DEPRECATED and the test execution was SKIPPED
  TEST PASSED!
> Executing 'TFM_PS_TEST_2003'
  Description: 'Get handle with null handle pointer (DEPRECATED)'
This test is DEPRECATED and the test execution was SKIPPED
  TEST PASSED!
> Executing 'TFM_PS_TEST_2004'
  Description: 'Get attributes interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2005'
  Description: 'Get attributes with null attributes struct pointer'
....
```

PSA API tests

Follow the build instructions for the PSA API tests and then follow the above procedures for flashing the image to the board. The PSA API tests are linked into the TF-M binaries and will automatically run. A log similar to the following should be visible on the UART; it is normal for some tests to be skipped but there should be no failed tests:

```
[Sec Thread] Secure image initializing!
Booting TFM v1.1
Non-Secure system starting...

***** PSA Architecture Test Suite - Version 1.0 *****

Running.. Storage Suite
******************************************


TEST: 401 | DESCRIPTION: UID not found check
[Info] Executing tests from non-secure

[Info] Executing ITS tests
[Check 1] Call get API for UID 6 which is not set
[Check 2] Call get_info API for UID 6 which is not set
[Check 3] Call remove API for UID 6 which is not set
[Check 4] Call get API for UID 6 which is removed
```

(continues on next page)

(continued from previous page)

```
[Check 5] Call get_info API for UID 6 which is removed
[Check 6] Call remove API for UID 6 which is removed
[Check 7] Call get API for different UID 5
[Check 8] Call get_info API for different UID 5
[Check 9] Call remove API for different UID 5

[Info] Executing PS tests
[Check 1] Call get API for UID 6 which is not set
[Check 2] Call get_info API for UID 6 which is not set
[Check 3] Call remove API for UID 6 which is not set
[Check 4] Call get API for UID 6 which is removed
[Check 5] Call get_info API for UID 6 which is removed
[Check 6] Call remove API for UID 6 which is removed
[Check 7] Call get API for different UID 5
[Check 8] Call get_info API for different UID 5
[Check 9] Call remove API for different UID 5

TEST RESULT: PASSED

******************************************

<further tests removed from log for brevity>

************ Storage Suite Report **********
TOTAL TESTS     : 17
TOTAL PASSED    : 11
TOTAL SIM ERROR : 0
TOTAL FAILED    : 0
TOTAL SKIPPED   : 6
******************************************
```

Note: The Internal Trusted Storage and Protected Storage flash areas must be wiped before running the Storage test suites.

Many IDEs for embedded development (such as Keil µVision) offer a function to erase a device's flash. Refer to your IDE's documentation for instructions.

Another way this can be achieved is by using `srec_cat` with the `-fill` parameter to fill the corresponding area in the binary with the erase value of the flash (`0xFF`).

Refer to the platform flash layout for appropriate addresses to erase on other platforms.

This step is not required on targets that emulate flash storage in RAM, as it will be erased each time the device is reset. Note, however, that a warm reset may not clear SRAM contents, so it may be necessary to power the device off and on again between test runs.

Example application or regression tests on Musca-B1 without BL2 bootloader

Follow the above procedures, but create a unified hex file out of `tfm_s.bin` and `tfm_ns.bin`:

- Windows:

```
srec_cat.exe bin\tfm_s.bin -Binary -offset 0xA000000 bin\tfm_ns.bin -Binary -offset␣
↪0xA080000 -o tfm.hex -Intel
```

- Linux:

```
srec_cat bin/tfm_s.bin -Binary -offset 0xA000000 bin/tfm_ns.bin -Binary -offset␣
↪0xA080000 -o tfm.hex -Intel
```

8.2.3 Execute TF-M example and regression tests on MPS3 boards

To run the example code on CoreLink SSE-200 Subsystem for MPS3 (AN524)

FPGA image is available to download here

To run BL2 bootloader and TF-M example application and tests in the MPS3 board, it is required to have SMM-SSE-200 for MPS3 (AN524) image in the MPS3 board SD card. The image should be located in `<MPS3 device name>/MB/HBI<BoardNumberBoardrevision>/AN524`

And the current boot memory for AN524 is QSPI flash, so you need to set the correct REMAP option in `<MPS3 device name>/MB/HBI<BoardNumberBoardrevision>/AN524/an524_v1.txt`

```
REMAP: QSPI                   ;REMAP boot device BRAM/QSPI.  Must match REMAPVAL below.
REMAPVAL: 1                   ;REMAP register value e.g. 0-BRAM. 1-QSPI
```

The MPS3 board tested is HBI0309B.

Note: If you change the exe names, MPS3 expects file names in 8.3 format.

Example application

1. Copy `bl2.bin` and `tfm_s_ns_signed.bin` files from build dir to `<MPS3 device name>/SOFTWARE/`
2. Open `<MPS3 device name>/MB/HBI0309B/AN524/images.txt`
3. Update the `images.txt` file as follows:

```
TITLE: Arm MPS3 FPGA prototyping board Images Configuration File

[IMAGES]
TOTALIMAGES: 2                     ;Number of Images (Max: 32)

IMAGE0UPDATE: AUTO                 ;Image Update:NONE/AUTO/FORCE
IMAGE0ADDRESS: 0x00000000          ;Please select the required executable program
IMAGE0FILE: \SOFTWARE\bl2.bin
IMAGE1UPDATE: AUTO
IMAGE1ADDRESS: 0x00040000
IMAGE1FILE: \SOFTWARE\tfm_s_ns_signed.bin
```

4. Close `<MPS3 device name>/MB/HBI0309B/AN524/images.txt`
5. Unmount/eject the `<MPS3 device name>` unit
6. Reset the board to execute the TF-M example application
7. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

   ```
   [INF] Starting bootloader
   [INF] Image 0: magic= good, copy_done=0xff, image_ok=0xff
   [INF] Scratch: magic=unset, copy_done=0x43, image_ok=0xff
   [INF] Boot source: slot 0
   [INF] Swap type: none
   [INF] Bootloader chainload address offset: 0x40000
   [INF] Jumping to the first image slot
   [Sec Thread] Secure image initializing!
   ```

Note: If `-DPLATFORM_DEFAULT_PROVISIONING=OFF` is set then the provisioning bundle has to be placed on the `0x001CB000` address.

Regression tests

After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[INF] Starting bootloader
[INF] Image 0: magic= good, copy_done=0xff, image_ok=0xff
[INF] Scratch: magic=unset, copy_done=0x9, image_ok=0xff
[INF] Boot source: slot 0
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x40000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!

#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_PS_TEST_2XXX)...
> Executing 'TFM_PS_TEST_2001'
  Description: 'Set interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2002'
  Description: 'Set interface with create flags'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2003'
  Description: 'Set interface with NULL data pointer'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2004'
  Description: 'Set interface with invalid data length'
  TEST PASSED!
....
```

Note: Some of the attestation tests take a few minutes to run on the MPS3.

Example application without BL2 bootloader

1. Copy `tfm_s.bin` and `tfm_ns.bin` files from build dir to `<MPS3 device name>/SOFTWARE/`
2. Open `<MPS3 device name>/MB/HBI0309B/AN524/images.txt`
3. Update the `images.txt` file as follows:

```
TITLE: Arm MPS3 FPGA prototyping board Images Configuration File

[IMAGES]
TOTALIMAGES: 2                     ;Number of Images (Max: 32)

IMAGE0UPDATE: AUTO                 ;Image Update:NONE/AUTO/FORCE
IMAGE0ADDRESS: 0x00000000          ;Please select the required executable program
IMAGE0FILE: \SOFTWARE\tfm_s.bin
IMAGE1UPDATE: AUTO
IMAGE1ADDRESS: 0x000C0000
IMAGE1FILE: \SOFTWARE\tfm_ns.bin
```

4. Close `<MPS3 device name>/MB/HBI0309B/AN521/images.txt`
5. Unmount/eject the `<MPS3 device name>` unit
6. Reset the board to execute the TF-M example application
7. After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[Sec Thread] Secure image initializing!
```

Regression tests without BL2 bootloader

After completing the procedure you should be able to visualize on the serial port (baud 115200 8n1) the following messages:

```
[Sec Thread] Secure image initializing!

#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_PS_TEST_2XXX)...
> Executing 'TFM_PS_TEST_2001'
  Description: 'Set interface'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2002'
  Description: 'Set interface with create flags'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2003'
  Description: 'Set interface with NULL data pointer'
  TEST PASSED!
> Executing 'TFM_PS_TEST_2004'
```

(continues on next page)

(continued from previous page)

```
  Description: 'Set interface with invalid data length'
  TEST PASSED!
....
```

Firmware upgrade and image validation with BL2 bootloader

High level operation of BL2 bootloader and instructions for testing firmware upgrade is described in *secure boot*.

8.3 Documentation generation

Two documents are available for generation:

- Reference Manual (HTML, PDF)
- User Guide (HTML, PDF)

The documentation build is independent from building the binary artifacts.

8.3.1 Tools and building environment

These tools are used to generate TF-M documentation:

- Doxygen v1.8.0 or later
- Graphviz dot v2.38.0 or later
- PlantUML v1.2018.11 or later
- Java runtime environment v1.8 or later (for running PlantUML)
- Sphinx and other python modules, listed in `docs/requirements.txt`

Additionally, for PDFs format:

- LaTeX
- PdfLaTeX

There are two ways of building TF-M reference manual:

1. As a custom target of TF-M CMake build system
2. Directly, using the command line tools

To prepare your building environment execute the following steps:

Linux

Windows

```
sudo apt-get install -y doxygen graphviz default-jre
mkdir ~/plantuml
curl -L http://sourceforge.net/projects/plantuml/files/plantuml.jar/download --output ~/
↪plantuml/plantuml.jar
export PLANTUML_JAR_PATH=~/plantuml/plantuml.jar

# For PDF generation
sudo apt-get install -y doxygen-latex

# Install the required Python modules
pip3 install --upgrade pip
cd trusted-firmware-m
pip3 install -r docs/requirements.txt
```

Download and install the following tools:

- Doxygen 1.8.8
- Graphviz 2.38
- **The Java runtime is part of the Arm DS installation or can be**
  downloaded from here
- PlantUML
- MikTeX - for PDF generation only

Set the environment variables, assuming that:

- doxygen, dot, and MikTeX binaries are available on the PATH.
- Java JVM is used from Arm DS installation.

```
set PLANTUML_JAR_PATH=<plantuml_Path>\plantuml.jar
set PATH=$PATH;<ARM_DS_PATH>\sw\java\bin

# Install the required Python modules
pip3 install --upgrade pip
cd trusted-firmware-m
pip3 install -r docs\requirements.txt
```

8.3.2 Build TF-M Reference Manual

The Reference Manual will be generated in the `build_docs/reference_manual`.

Linux

Windows

```
cd <TF-M base folder>
cmake -S docs -B build_docs
cmake --build build_docs -- tfm_docs_refman_html tfm_docs_refman_pdf
```

```
cd <TF-M base folder>
cmake -S docs -B build_docs -G"Unix Makefiles"
cmake --build build_docs -- tfm_docs_refman_html tfm_docs_refman_pdf
```

8.3.3 Build TF-M User Guide

The User Manual will be available under the directory `build_docs/user_guide`.

Linux

Windows

```
cd <TF-M base folder>
cmake -S docs -B build_docs
cmake --build build_docs -- tfm_docs_userguide_html tfm_docs_userguide_pdf
```

```
cd <TF-M base folder>
cmake -S docs -B build_docs -G"Unix Makefiles"
cmake --build build_docs -- tfm_docs_userguide_html tfm_docs_userguide_pdf
```

8.3.4 Direct build using a command line tools

The direct build will build both user_guide and reference_manual.

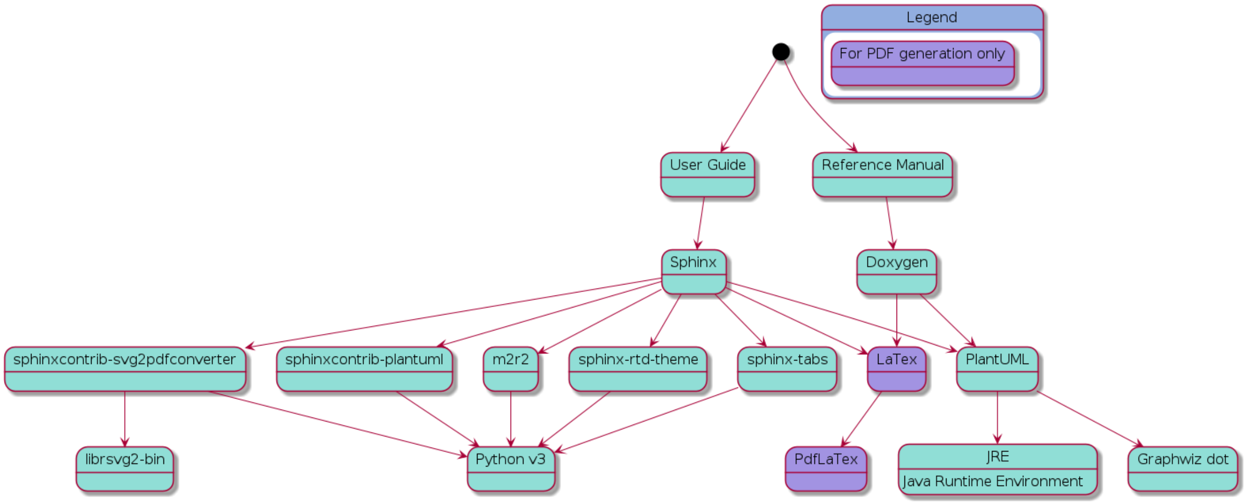
Linux

Windows

```
# Build the documentation from build_docs directory
cd <TF-M base folder>
mkdir build_docs
cp docs/conf.py build_docs/conf.py
cd build_docs
sphinx-build ./ user_guide
```

```
# Command line tools is yet not available for Windows
```

8.3.5 Dependencies



8.4 Additional build instructions for the IAR toolchain

Follow the instructions in *software requirements*, but replace the -DTFM_TOOLCHAIN_FILE setting with toolchain_IARARM.cmake.

8.4.1 Notes for building with IARARM

> IAR Embedded Workbench for ARM (EWARM) versions 8.42 or later are required.
>
> Currently the MUSCA_B1 and MUSCA_S1 targets are not supported with IARARM, due to lack of testing.
>
> cmake needs to be version 3.14 or newer.
>
> The V8M IAR CMSIS_5 RTX libraries in CMSIS_5 5.5.0 has a problem and has been updated in CMSIS_5 5.7.0. The updated libraries are part of the tf-m-tests repo and no special instructions are needed when the libraries from this repo are used.
>
> For all configurations and build options some of the QCBOR tests fail due to the tests not handling double float NaN:s according to the Arm Runtime ABI. This should be sorted out in the future.

Example: building TF-M for AN521 platform using IAR:

```
cd <TF-M base folder>
cmake -S . -B cmake_build -DTFM_PLATFORM=arm/mps2/an521 -DTFM_TOOLCHAIN_FILE=toolchain_
↪IARARM.cmake
cmake --build cmake_build -- install
```

Alternately using traditional cmake syntax

```
cd <TF-M base folder>
mkdir cmake_build
cd cmake_build
cmake .. -DTFM_PLATFORM=arm/mps2/an521 -DTFM_TOOLCHAIN_FILE=../toolchain_IARARM.cmake
make install
```

Regression Tests for the AN521 target platform

```
cd <TF-M base folder>
cmake -S . -B cmake_build -DTFM_PLATFORM=arm/mps2/an521 -DTFM_TOOLCHAIN_FILE=toolchain_
↪IARARM.cmake -DTEST_S=ON -DTEST_NS=ON
cmake --build cmake_build -- install
```

Alternately using traditional cmake syntax

```
cd <TF-M base folder>
mkdir cmake_build
cd cmake_build
cmake .. -DTFM_PLATFORM=arm/mps2/an521 -DTFM_TOOLCHAIN_FILE=../toolchain_IARARM.cmake -
↪DTEST_S=ON -DTEST_NS=ON
make install
```

8.5 Building TF-M (SPE)

This build generates the SPE binary and artifacts, necessary for *Building Application (NSPE)*.

8.5.1 Getting the source code

```
cd <base folder>
git clone https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git
```

In this documentation, the cloned `trusted-firmware-m` repository will be referenced as `<TF-M source dir>`. Additionally, TF-M depends on several external projects, handled by CMake automatically but you can alter that behaviour using *Dependency management*.

Note:

- For building with Armclang compiler version 6.10.0+, please follow the note in TF-M getting started.
- For building with the IAR toolchain, please see the notes in *IAR software requirements*
- Please use "/" instead of "\" for paths when running CMAKE commands under Windows Command Prompt.

8.5.2 Configuring

TF-M has many config options for configuring and fine-tuning. Please check the *Configuration* section for the details. The **base** (default) configuration contains only essential components such as SPM and HW platform support hence the only mandatory argument to TF-M build is a platform name, provided via CMake command-line option `-DTFM_PLATFORM=<platform name>`, it can be:

- A relative path under `<TF-M source dir>/platform/ext/target`, for example `arm/mps2/an521`.
- An absolute path of target platform, mainly used for out-of-tree platform build.
- A basename of the target platform folder, for example `an521`.

Essential Directories

There are 3 essential directories used by CMake for building TF-M:

- Source code directory `<TF-M source dir>`
- Build directory `<Build Dir>` - the location of all intermediate files required to produce a build target.
- Install directory `<Artifact Dir>` - the location of the build output files.

Note::
It's recommended to use absolute paths for all directories. Relative paths may not fully work.

Toolchains

TF-M supports 3 toolchains for cross-compiling and building the project binaries:

- GNU - **default**
- ArmClang
- IAR

Each toolchain has a configuration file for the compiler and linker. They are located at the root directory of TF-M. Use `TFM_TOOLCHAIN_FILE` option to provide the absolute path to the preferred toolchain file, or relative path to working directory. The default **toolchain_GNUARM.cmake** is selected by *config_base.cmake* file if the option is omitted.

Build type

By default, a MinSizeRel configuration is built. Alternate build types can be specified with the `CMAKE_BUILD_TYPE` variable. The possible types are:

- Debug
- RelWithDebInfo
- Release
- MinSizeRel - **default**

Debug symbols are added by default to all builds, but can be removed from *Release* and *MinSizeRel* builds by setting `TFM_DEBUG_SYMBOLS` to *OFF*.

RelWithDebInfo, *Release* and *MinSizeRel* all have different optimizations turned on and hence will produce smaller, faster code than *Debug*. *MinSizeRel* will produce the smallest code and hence is often a good idea on RAM or flash-constrained systems.

Output files

In a successful build, a set of files will be created in the `<Artifact Dir>`. By default, it is `<Build Dir>\api_ns` subfolder but you can redirect the output to any location using `CMAKE_INSTALL_PREFIX` option. It can be an absolute path or relative to your current directory. For the contents of the artifact directory please refer to *SPE artifacts structure*.

Other build parameters

The full list of default options is in `config/config_base.cmake` and explained in *Build configuration*. Several important options are listed below.

Parameter	Description	Default value
BL2	Build level 2 secure bootloader.	ON
PROJECT_CONFIG_HEADER_FILE	User defined header file for TF-M config	
TFM_ISOLATION_LEVEL	Set TF-M isolation level.	1
TFM_PROFILE	See *TF-M Profiles*.	

Project Config Header File

CMake variable `PROJECT_CONFIG_HEADER_FILE` can be set by a user the full path to a configuration header file, which is used to fine-tune component options. The detailed reference for the project config header file is in *The Header File Config System*.

8.5.3 Building binaries

The command below shows a general template for building TF-M as a typical CMake project:

```
cmake -S <TF-M source dir> -B <Build Dir> -DTFM_PLATFORM=<platform>
cmake --build <Build Dir> -- install
```

Note: It is recommended to clean up the build directory before re-build if the config header file is updated. CMake is unable to automatically recognize the dependency when the header file is defined as a macro.

Building default configuration for an521

```
cd <TF-M source dir>
cmake -S . -B build -DTFM_PLATFORM=arm/mps2/an521
cmake --build build -- install
```

The command above is intended to do:

- take TF-M sources in the current `.` folder
- build SPE in the `build` folder
- for **an521** platform
- using GNU toolchain *by default*. Use `-DTFM_TOOLCHAIN_FILE=<toolchain file>` for alternatives as described in *Toolchains*
- install output files in `build/api_ns` folder *by default*. You can specify a different directory using `-DCMAKE_INSTALL_PREFIX=<Artifact dir>` as described in *Output files*

Note: It is recommended to build each different build configuration in a separate build directory.

CMake can generate code for many native build systems. TF-M is tested with `Unix Makefiles` (default) and `Ninja`. The `-G` option can specify alternative generators. For example for building with Ninja in the Debug *Build type* using ArmClang *Toolchains* you can use the following:

```
cd <TF-M source dir>
cmake -S . -B build -DTFM_PLATFORM=arm/mps2/an521 -GNinja -DTFM_TOOLCHAIN_FILE=toolchain_
↪ARMCLANG.cmake -DCMAKE_BUILD_TYPE=Debug
cmake --build build -- install
```

8.5.4 Dependency management

The TF-M build system will fetch all dependencies by default with appropriate versions and store them inside the build tree. In this case, the build tree location is `<build_dir>/lib/ext`.

If you have local copies already and wish to avoid having the libraries downloaded every time the build directory is deleted, then the following variables can be set to the paths to the root directories of the local repos. This will disable the automatic downloading for that dependencies and speed up development iterations or allow usage of a dependency version different from the current one. Additionally, these path variables can be set in `localrepos.cmake` file which will be included in a build if it exists. This file is ignored in TF-M git settings.

The following table lists the commonly used repos. For others, you can refer to `lib/ext`.

Dependency	Cmake variable	Git repo URL
Mbed Crypto	MBEDCRYPTO_PATH	https://github.com/ARMmbed/mbedtls
MCUboot	MCUBOOT_PATH	https://github.com/mcu-tools/mcuboot
QCBOR	QCBOR_PATH	https://github.com/laurencelundblade/QCBOR.git

The recommended versions of the dependencies are listed in `config/config_base.cmake`.

Note:

- Some repositories might need patches to allow building it as a part of TF-M. While these patches are being upstreamed they are stored in a dependency folder under `lib/ext/`. In order to use local repositories those patches shall be applied to original source. An alternative is to copy out the auto-downloaded repos under the `<build_dir>/lib/ext`. They have been applied with patches and can be used directly.

Example: building TF-M with local Mbed Crypto repo

Preparing a local repository consists of 2 steps: cloning and patching. This is only required to be done once. For dependencies without `.patch` files in their `lib/ext` directory the only required step is cloning the repo and checking out the correct branch.

```
cd <Mbed Crypto base folder>
git clone https://github.com/ARMmbed/mbedtls
cd mbedtls
git checkout <MBEDCRYPTO_VERSION> from <TF-M source dir>/config/config_base.cmake>
git apply <TF-M source dir>/lib/ext/mbedcrypto/*.patch
```

Note: `<Mbed Crypto base folder>` does not need to have any fixed position related to the TF-M repo so alternative method to get prepared dependency repos is to let TF-M download it once and then copy them out of the `build/lib/ext` folder.

Now build TF-M binaries

```
cd <TF-M source dir>
cmake -S . -B build -DTFM_PLATFORM=arm/mps2/an521 -DMBEDCRYPTO_PATH=<Mbed Crypto base␣
↪folder>/mbedtls
cmake --build build -- install
```

8.6 Building Application (NSPE)

As a result of *Building TF-M (SPE)* you will get a set of *Output files* in `<Artifact Dir>` required for building TF-M application. Essentially, SPE exports a binary and a set of C source files for PSA interface and platform. Please note that NSPE and SPE are independent projects and can be built using different toolchains and toolchain options.

8.6.1 SPE artifacts structure

SPE components prepared and installed for NSPE usage in `<Artifact Dir>` will have the following structure:

```
<Artifact Dir>
├── bin
├── cmake
├── config
├── image_signing
├── interface
├── platform
└── CMakeLists.txt
```

With certain configurations, additional folders may also be installed. These folders have the following content:

- **bin** - binary images of SPE, Bootloader(optional) and combined.
- **cmake** - CMake scripts like SPE configuration and *NSPE toolchains*.
- **config** - Configuration files
- **image_signing** - binary image signing tool and keys.
- **interface** - PSA interface exposed by SPE.
- **platform** - source code for a selected hardware platform.
- **CMakeLists.txt** - CMake script for the artifacts integration in NSPE.

The content of `<Artifact Dir>` is an exported directory for integration with CMake projects.

Note: Attempting to change any file in <Artifact Dir> may cause incompatibility issues. Instead, please change the corresponding file in the <TF-M source dir>.

8.6.2 NSPE toolchains

SPE prepares and exports CMake toolchain files for building NSPE in all supported *Toolchains* in `<Artifact Dir>/cmake` folder. Toolchain used to build NSPE can be different from what is used to build SPE.

8.6.3 Basic SPE integration

Refer to the example of TF-M applications in **tf-m-extras** repository.

CHAPTER NINE

CONFIGURATION

9.1 Build configuration

All configuration options are provided by cmake variables, and their default values, with docstrings, can be found in `config/config_base.cmake`.

Configuration is provided in multiple stages. Each stage will not override any config that has already been set at any of the prior stages.

1. Command-line variable settings are applied.
2. If the `TFM_EXTRA_CONFIG_PATH` variable has been set, that file is loaded.
3. If TEST_PSA_TEST is set, then PSA API test related config is applied from `config/tests/config_test_psa_api.cmake`.
4. If it exists, CMAKE_BUILD_TYPE specific config is applied from `config/build_type/<build_type>.cmake`.
5. Target specific config from `platform/ext/target/<target_platform>/config.cmake` is applied.
6. If CRYPTO_HW_ACCELERATOR is set, then a config specific to the accelerator type is applied if it exists.
7. If it exists, TFM Profile specific config is applied from `config/profile/<tfm_profile>.cmake`.
8. `config/config_default.cmake` is loaded.
9. If `TEST_S` or `TEST_NS` or other single test suite config like `TEST_NS_ATTESTATION` (see *Tests configuration*) is set, then config from `${TFM_TEST_REPO_PATH}/test/config/set_config.cmake` and `${TFM_TEST_REPO_PATH}/test/config/default_ns_test_config.cmake` or `${TFM_TEST_REPO_PATH}/test/config/default_s_test_config.cmake` or `${TFM_TEST_REPO_PATH}/test/config/default_test_config.cmake` is applied.

> **Warning:** This means that command-line settings are not applied when they conflict with required platform settings. If it is required to override platform settings (this is not usually a good idea) then TFM_EXTRA_CONFIG_PATH should be used.

9.2 The Header File Config System

The header file configurations system is used to fine-tune component options.

The following diagram shows how the system works.

Source files shall include `config_tfm.h` when necessary to fetch Component option settings. It is expected that all Component options are included in `config_tfm.h` to explicitly set values for each option.

The `config_tfm.h` includes base configuration `config_base.h`. Refer to *Base Configuration* for details of the base configurations.

The `config_tfm.h` includes a customized project config file provided via compile definition `PROJECT_CONFIG_HEADER_FILE`. Customized Component options in the project config file overrides those configured in `config_base.h`. The project config header file can be

- Generated by the TF-M Kconfig system <kconfig_system>
- One of the header files of Profiles <tf-m_profiles>, set via the `TFM_PROFILE` build option.
- Manually customized profile based on pre-set profiles.

Users set CMake variable `PROJECT_CONFIG_HEADER_FILE` with the full path of the configuration header file.

A platform can adjust or place restriction on config options by providing a `config_tfm_target.h` under the root folder of their platforms. If the build system finds the file, it sets the `TARGET_CONFIG_HEADER_FILE` compile definition. Platform specific option settings in `TARGET_CONFIG_HEADER_FILE` overrides those configured in `config_base.h`.

```
#ifdef TARGET_CONFIG_HEADER_FILE
#include TARGET_CONFIG_HEADER_FILE
#endif

#ifdef PROJECT_CONFIG_HEADER_FILE
#include PROJECT_CONFIG_HEADER_FILE
#endif

#include "config_base.h"
```

Each component can implement a `config_<comp_name>_check.h` to validate component specific config dependencies. `config_<comp_name>_check.h` shall include `config_tfm.h`

9.3 The Kconfig System

The Kconfig system is an alternative tool to the CMake config system for users to change config options of TF-M.

It handles dependencies and validations automatically when you change configurations so that the generated configuration options are always valid.

To use the Kconfig system, enable `USE_KCONFIG_TOOL` in command line. And enable `MENUCONFIG` to launch configuration GUI.

The Kconfig system consists of *The Kconfig tool* and the *The Kconfig files*.

9.3.1 The Kconfig tool

The Kconfig tool is a python script based on Kconfiglib to generate the following config files:

- CMake config file

  Contains CMake cache variables of building options.

- Header file

  Contains component options in the header file system. Component options are gathered together in a separate menu `TF-M component configs` in *The Kconfig files*.

- The .config and .config.old files

  The `.config` file which contains all the above configurations in the Kconfig format. It will be created after the first execution of the script. It is only used to allow users to make adjustments basing on the previous settings. The Kconfig tool will load it if it exists and `.config.old` will be created to save the previous configurations.

The tool supports loading multiple pre-set configuration files merging into a single one. The first loaded options are overridden by later ones if the config files contain duplicated options. And dependencies between config options are taken care of. It then launches a configuration GUI for users to change any config options if the `MENUCONFIG` is enabled in build command line.

Integration with TF-M build system

TF-M build system includes `kconfig.cmake` to integrate this tool. It prepares the parameters for the script and invokes it to load multiple configuration files basing on your build setup, including but not limited to

- Build type bound configurations, decided by `CMAKE_BUILD_TYPE`
- Profile configurations, decided by `TFM_PROFILE`

9.3.2 Customizing config options

By default, the Kconfig system only merges configuration files and generated the final config files. To customize the config options, there are several approaches.

Menuconfig

Menuconfig is the recommended approach to adjust the values of the config options because it has a graphic interface for you to easily change the options without worrying about dependencies.

To launch the menuconfig, you need to enable `MENUCONFIG` in addition to enabling `USE_KCONFIG_TOOL`.

```
cmake -S . -B cmake_build -DTFM_PLATFORM=arm/mps2/an521 \
                          -DUSE_KCONFIG_TOOL=ON \
                          -DMENUCONFIG=ON
```

Note: Although the Kconfiglib provides three menuconfig interfaces, only GUI menuconfig can be launched by CMake for the time being.

Command line options

The support of passing configurations via command line is kept for the Kconfig system.

```
cmake -S . -B cmake_build -DTFM_PLATFORM=arm/mps2/an521 \
                          -DUSE_KCONFIG_TOOL=ON \
                          -DTFM_ISOLATION_LEVEL=2
```

Kconfig file

You can also put the frequently used config options into a Kconfig file. When you need to apply the config options in that file, pass it via command line option `-DKCONFIG_CONFIG_FILE`

```
cmake -S . -B cmake_build -DTFM_PLATFORM=arm/mps2/an521 \
                          -DTFM_ISOLATION_LEVEL=2 \
                          -DUSE_KCONFIG_TOOL=ON \
                          -DKCONFIG_CONFIG_FILE=my_config.conf
```

Note: The command line set options override the ones in the config file. And you can always launch menuconfig to do the final adjustments.

9.3.3 The Kconfig files

The Kconfig files are the files written by the Kconfig language to describe config options. They also uses some Kconfiglib extensions such as optional source `osource` and relative source `rsource` so they can only work with the Kconfiglib.

9.4 TF-M Profiles

The capabilities and resources may dramatically vary on different IoT devices. Some IoT devices may have very limited memory resource. The program on those devices should keep small memory footprint and basic functionalities. On the other hand, some devices may consist of more memory and extended storage, to support stronger software capabilities.

Diverse IoT use cases also require different levels of security and requirements on device resource. For example, use cases require different cipher capabilities. Selecting cipher suites can be sensitive to memory footprint on devices with constrained resource.

Trusted Firmware-M (TF-M) defines several general profiles, such as Profile Small, Profile Medium, Profile Medium ARoT-less and Profile Large, to provide different levels of security to fit diverse device capabilities and use cases applied on the top of the base configuration.

Each profile specifies a predefined list of features, targeting typical use cases with specific hardware constraints. Profiles can serve as reference designs, based on which developers can continue further development and configurations, according to use case.

TF-M Profiles align with Platform Security Architecture specifications and certification guidelines. It can help vendors to simplify security configuring for PSA certification.

Please check the table below to compare differences while details are discussed in the links below.

9.4.1 Trusted Firmware-M Profile Small Design

Introduction

As one of the TF-M Profiles, TF-M Profile Small (Profile S) consists of lightweight TF-M framework and basic Secure Services to keep smallest memory footprint, supporting fundamental security features on devices with ultra constrained resource.

This profile enables connecting with Edge Gateways and IoT Cloud Services supporting secure connection based solely on symmetric cryptography.

This document summarizes and discusses the features specified in TF-M Profile Small.

Overall design

TF-M Profile Small defines the following features:

- Lightweight framework
  - Secure Function (SFN) model[2]
  - Level 1 isolation
  - Buffer sharing allowed
  - Single secure context
- Crypto
  - Symmetric cipher only
  - Cipher suite for symmetric-key algorithms based protocols, such as cipher suites defined in TLS pre-shared key (TLS-PSK)[1].
    - Advanced Encryption Standard (AES) as symmetric crypto algorithm
    - SHA256 as Hash function
    - HMAC as Message Authentication Code algorithm
  - Only enable multi-part functions in hash, symmetric ciphers, Message Authentication Code (MAC) and Authenticated Encryption with Associated Data (AEAD) operations.
- Internal Trusted Storage (ITS)
  - No encryption
  - No rollback protection
  - Decrease internal transient buffer size
- Initial Attestation
  - Based on symmetric key algorithms
- Lightweight boot
  - Single image boot
  - Anti-rollback protection is enabled

Protected Storage, firmware update and other Secure Services provided by TF-M are disabled by default.

[2] Arm Firmware Framework for M 1.1 Extensions

[1] Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)

Design details

More details of TF-M Profile Small design are discussed in following sections.

Lightweight framework

TF-M framework model

SFN model is selected by default in Profile Small implementation. SFN model is defined in FF-M 1.1 extensions[Page 151, 2]. It is a more simple implementation of TF-M framework and may also reduce memory footprint, compared with Inter-Process Communication (IPC) model[3].

Level 1 isolation

PSA Security Model[4] defines 3 levels of isolation.

- Level 1 isolation isolates Secure Processing Environment (SPE) from Non-secure Processing Environment (NSPE).
- PSA Root of Trust (PSA RoT) and Application Root of Trust (ARoT) are isolated from each other in level 2 isolation.
- Individual secure partitions are isolated from each other even within a particular security domain (PSA RoT, ARoT), in level 3 isolation.

Profile Small dedicated use cases with simple service model may not require level 2 or level 3 isolation. Devices which Profile Small aims at may be unable to implement stricter isolation, limited by hardware capabilities.

Level 1 isolation reduces requirements enforced by hardware isolation and cost of software for management.

Note: Security note

If a device or a use case enforces level 2 or level 3 isolation, it is suggested to apply other configurations, other than TF-M Profile Small.

Crypto service

TF-M Profile Small only requires symmetric crypto since symmetric algorithms require shorter keys and less computational burden, compared with asymmetric crypto.

By default, TF-M Profile Small requires the same capabilities as defined in TLS-PSK, to support symmetric key algorithms based protocols.

Note: Implementation note

Please note that TF-M Profile Small doesn't require that TLS-PSK is mandatory in applications. Instead, Profile Small only requires the same capabilities as defined in TLS-PSK, such as one symmetric cipher algorithm and one hash function.

[3] Arm Platform Security Architecture Firmware Framework 1.0

[4] Platform Security Model 1.1

TF-M Profile Small selects TLS-PSK cipher suite TLS_PSK_WITH_AES_128_CCM[5] as reference, which requires:

- AES-128-CCM (AES CCM mode with 128-bit key) as symmetric crypto algorithm
- SHA256 as Hash function
- HMAC as Message Authentication Code algorithm

TLS_PSK_WITH_AES_128_CCM is selected since it requires small key length and less hardware capabilities, while keeping enough level of security.

Note: Implementation note

Developers can replace default algorithms with others or implement more algorithms.

Proper symmetric key algorithms and cipher suites should be selected according to device capabilities, the use case and the requirement of peers in connection.

Refer to *Crypto service configuration* for implementation details of configuring algorithms and cipher suites.

Note: Security note

It is recommended not to use MD5 or SHA-1 for message digests as they are subject to collision attacks[6][7].

By default, Profile Small only enables multi-part functions defined in PSA Cryptography API[13] in hash, symmetric ciphers, MAC and AEAD operations. Disabling single-part functions optimizes the code size of TF-M crypto service. Multi-part operations allows the message data to be processed in fragments instead of all at once. In static memory allocation, single-part operation may require to allocate a large memory space to support long message with unknown length. Therefore single-part operations can help users optimize memory footprint, especially while dealing with streaming data on IoT devices.

It may slightly increase the code size in applications to replace single-part implementation with mulit-part implementation. Althgouth the code size increment can be qaulified, if users are concerned about the code size increment, they can enable single-part operations by toggling Profile Small default configuration.

It may increase latency and overall time cost to implement cryptography functionality with single-part operations, compared to with multi-part ones. Users can enable single-part operations if the usage scenario requires single-part opreations to meet its perfermance metrics.

Secure Storage

TF-M Profile Small assumes that extremely constrained devices only contain basic on-chip storage, without external or removable storage. As a result, TF-M Profile Small includes ITS service and disables Protected Storage service.

[5] AES-CCM Cipher Suites for Transport Layer Security (TLS)

[6] Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms

[7] Transitioning the Use of Cryptographic Algorithms and Key Lengths

[13] PSA Cryptography API 1.0

Encryption and rollback protection

Neither encryption nor rollback protection is enabled in current ITS implementation.

It is expected that ITS relies solely on the physical inaccessibility property of on-chip storage, together with PSA isolation, without requiring additional cryptographic protection.

Internal transient buffer

ITS implements a internal transient buffer[8] to hold the data read from/written to storage, especially for flash, to solve the alignment and security issues.

The internal transient buffer is aligned to the flash device's program unit. Copying data to it from the caller can align all write requests to the flash device's program unit. The internal transient buffer can help protect Flash access from some attacks, such as TOCTOU attack.

Although removing this internal buffer can save some memory consumption, typically 512 bytes, it may bring alignment or security issues. Therefore, to achieve a better trade-off between memory footprint and security, TF-M Profile Small optimizes the internal buffer size to 32 bytes by default.

As discussed in *Crypto service*, TF-M Profile Small requires AES-128 and SHA-256, which use 128-bit key and 256-bit key respectively. Besides, either long public/private keys or PKI-based certificates should be very rare as asymmetric crypto is not supported in Profile Small. Therefore, a 32-byte internal buffer should cover the assets in TF-M Profile Small use cases.

The buffer size can be adjusted according to use case and device Flash attributes. Refer to *Internal Trusted Storage configurations* for more details.

Initial Attestation

Profile Small requires an Initial Attestation secure service based on symmetric key algorithms. Refer to PSA Attestation API document[9] for details of Initial Attestation based on symmetric key algorithms.

It can heavily increase memory footprint to support Initial Attestation based on asymmetric key algorithms, due to asymmetric ciphers and related PKI modules.

Note: Implementation note

As pointed out by PSA Attestation API document[Page 154, 9], the use cases of Initial Attestation based on symmetric key algorithms can be limited due to the associated infrastructure costs for key management and operational complexities. It may also restrict the ability to interoperate with scenarios that involve third parties.

If asymmetric key algorithms based Initial Attestation is required in use scenarios, it is recommended to select other TF-M Profiles which support asymmetric key algorithms.

Note: Implementation note

It is recommended to utilize the same MAC algorithm supported in Crypto service to complete the signing in `COSE_Mac0`, to minimize memory footprint.

[8] *ITS integration guide*

[9] PSA Attestation API 1.0 (ARM IHI 0085)

Lightweight boot

If MCUBoot provided by TF-M is enabled, single image boot[10] is selected by default in Profile Small. In case of single image boot, secure and non-secure images are handled as a single blob and signed together during image generation.

However, secure and non-secure images must be updated together in single image boot. It may decrease the flexibility of image update and cost longer update process. Since the image sizes should usually be small with limited functionalities in Profile Small dedicated use case, the cost may still be reasonable.

BL2 implementation can be device specific. Devices may implement diverse boot processes with different features and configurations. However, anti-rollback protection is required as a mandatory feature of boot loader. Boot loader should be able to prevent unauthorized rollback, to protect devices from being downgraded to earlier versions with known vulnerabilities.

Implementation

Overview

The basic idea is to add dedicated profile CMake configuration files under folder `config/profile` for TF-M Profile Small default configuration.

The top-level Profile Small config file collects all the necessary configuration flags and set them to default values, to explicitly enable the features required in Profile Small and disable the unnecessary ones, during TF-M build.

A platform/use case can provide a configuration extension file to overwrite Profile Small default setting and append other configurations. This configuration extension file can be added via parameter `TFM_EXTRA_CONFIG_PATH` in build command line.

The behavior of the Profile Small build flow (particularly the order of configuration loading and overriding) can be found at *Build configuration*

The details of configurations will be covered in each module in *Implementation details*.

Implementation details

This section discusses the details of Profile Small implementation.

Top-level configuration files

The firmware framework configurations in `config/profile/profile_small` are shown below.

[10] *Secure boot*

Table 28:: TFM options in Profile Small top-level CMake config file

Configs	Default value	Descriptions
TFM_ISOLATION_LEVEL	1	Select level 2 isolation
TFM_PARTITION_INTERNAL_TRUSTED_STORAGE	ON	Enable ITS SP
ITS_BUF_SIZE	32	ITS internal transient buffer size
TFM_PARTITION_CRYPTO	ON	Enable Crypto service
TFM_MBEDCRYPTO_CONFIG_PATH	${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/tfm_mbedcrypto_config_profile_small.h	Mbed Crypto config file path
TFM_MBEDCRYPTO_PSA_CRYPTO_CONFIG_PATH	${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/crypto_config_profile_small.h	Mbed Crypto PSA config file path
CRYPTO_ASYM_SIGN_MODULE_ENABLED	OFF	Enable asymmetric signature
CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED	OFF	Enable asymmetric encryption
TFM_PARTITION_INITIAL_ATTESTATION	ON	Enable Initial Attestation service
SYMMETRIC_INITIAL_ATTESTATION	ON	Enable symmetric attestation
TFM_PARTITION_PROTECTED_STORAGE	OFF	Enable PS service
TFM_PARTITION_PLATFORM	OFF	Enable TF-M Platform SP

Note: Implementation note

The following sections focus on the feature selection via configuration setting. Dedicated optimization on memory footprint is not covered in this document.

Device configuration extension

To change default configurations and add platform specific configurations, a platform can add a platform configuration file at `platform/ext<TFM_PLATFORM>/config.cmake`

TF-M framework setting

The top-level Profile Small CMake config file selects SFN model and level 1 isolation.

In SFN model, `-DPSA_FRAMEWORK_HAS_MM_IOVEC` is enabled by default. It reduces memory footprint by avoiding the transient copy from input vectors and copy to output vectors.

Crypto service configuration

Crypto Secure Partition

TF-M Profile Small enables Crypto Secure Partition (SP) in its top-level CMake config file. Crypto SP modules not supported in TF-M Profile Small are disabled. The disabled modules/features are shown below.

- Disable asymmetric cipher
- Disable single-part operations in Hash, MAC, AEAD and symmetric ciphers via selecting `CRYPTO_SINGLE_PART_FUNCS_DISABLED`

Other modules and configurations[11] are kept as default values.

Additional configuration flags with more fine granularity can be added to control building of specific crypto algorithms and corresponding test cases.

Mbed Crypto configurations

TF-M Profile Small adds a dedicated Mbed Crypto config file `tfm_mbedcrypto_config_profile_small.h` and Mbed Crypto PSA config file `crypto_config_profile_small.h` at `/lib/ext/mbedcrypto/mbedcrypto_config` folder, instead of the common one `tfm_mbedcrypto_config_default.h` and `crypto_config_default.h`[Page 157, 11].

Major Mbed Crypto configurations are set as listed below:

- Enable SHA256
- Enable generic message digest wrappers
- Enable AES
- Enable CCM mode for symmetric ciphers
- Disable other modes for symmetric ciphers
- Disable asymmetric ciphers
- Disable HMAC-based key derivation function (HKDF)

Other configurations can be selected to optimize the memory footprint of Crypto module.

A device/use case can append an extra config header to the Profile Small default Mbed Crypto config file. This can be done by setting the `TFM_MBEDCRYPTO_PLATFORM_EXTRA_CONFIG_PATH` cmake variable in the platform config file `platform/ext<TFM_PLATFORM>/config.cmake`. This cmake variable is a wrapper around the `MBEDTLS_USER_CONFIG_FILE` options, but is preferred as it keeps all configuration in cmake.

Internal Trusted Storage configurations

ITS service is enabled in top-level Profile Small CMake config file.

The internal transient buffer size `ITS_BUF_SIZE`[Page 154, 8] is set to 32 bytes by default. A platform/use case can overwrite the buffer size in its specific configuration extension according to its actual requirement of assets and Flash attributes.

Profile Small CMake config file won't touch the configurations of device specific Flash hardware attributes[Page 154, 8].

[11] *Crypto design*

Initial Attestation secure service

TF-M Profile Small provides a reference implementation of symmetric key algorithms based Initial Attestation, using HMAC SHA-256 as MAC algorithm in `COSE_Mac0` structure. The implementation follows PSA Attestation API document[Page 154, 9].

Profile Small top-level config file enables Initial Attestation secure service and selects symmetric key algorithms based Initial Attestation by default.

- Set `TFM_PARTITION_INITIAL_ATTESTATION` to `ON`
- Set `SYMMETRIC_INITIAL_ATTESTATION` to `ON`

Symmetric and asymmetric key algorithms based Initial Attestation can share the same generations of token claims, except Instance ID claim.

Profile Small may implement the procedure or rely on a 3rd-party tool to construct and sign `COSE_Mac0` structure.

Details of symmetric key algorithms based Initial Attestation design will be covered in a dedicated document.

Disabled secure services

Protected Storage and Platform Service are disabled by default in Profile Small top-level CMake config file.

Test configuration

Some cryptography tests are disabled due to the reduced Mbed Crypto config. Some of them are shown in the table below.

Table 29:: TFM options in Profile Small top-level CMake config file

Configs	Default value	Descriptions
`TFM_CRYPTO_TEST_ALG_CBC`	`OFF`	Test CBC cryptography mode
`TFM_CRYPTO_TEST_ALG_CCM`	`ON`	Test CCM cryptography mode
`TFM_CRYPTO_TEST_ALG_CFB`	`OFF`	Test CFB cryptography mode
`TFM_CRYPTO_TEST_ALG_ECB`	`OFF`	Test ECB cryptography mode
`TFM_CRYPTO_TEST_ALG_CTR`	`OFF`	Test CTR cryptography mode
`TFM_CRYPTO_TEST_ALG_OFB`	`OFF`	Test OFB cryptography mode
`TFM_CRYPTO_TEST_ALG_GCM`	`OFF`	Test GCM cryptography mode
`TFM_CRYPTO_TEST_ALG_SHA_384`	`OFF`	Test SHA-384 cryptography algorithm
`TFM_CRYPTO_TEST_ALG_SHA_512`	`OFF`	Test SHA-512 cryptography algorithm
`TFM_CRYPTO_TEST_HKDF`	`OFF`	Test HKDF key derivation algorithm
`TFM_CRYPTO_TEST_ECDH`	`OFF`	Test ECDH key agreement algorithm
`TFM_CRYPTO_TEST_CHACHA20`	`OFF`	Test ChaCha20 stream cipher
`TFM_CRYPTO_TEST_CHACHA20_POLY1305`	`OFF`	Test ChaCha20-Poly1305 AEAD algorithm
`TFM_CRYPTO_TEST_SINGLE_PART_FUNCS`	`OFF`	Test single-part operations in hash, MAC, AEAD and symmetric ciphers

BL2 setting

Profile Small enables MCUBoot provided by TF-M by default. A platform can overwrite this configuration by disabling MCUBoot in its configuration extension file `platform/ext<TFM_PLATFORM>/config.cmake`.

If MCUBoot provided by TF-M is enabled, single image boot is selected in TF-M Profile Small top-level CMake config file.

If a device implements its own boot loader, the configurations are implementation defined.

Table 30:: BL2 options in Profile Small top-level CMake config file

Configs	Default value	Descriptions
`BL2`	`ON`	Enable MCUBoot bootloader
`MCUBOOT_IMAGE_NUMBER`	`1`	Combine S and NS images

Platform support

Building Profile Small

To build Profile Small, argument `TFM_PROFILE` in build command line should be set to `profile_small`.

Take AN521 as an example.

The following commands build Profile Small without test cases on **AN521** with build type **MinSizeRel**, built by **Armclang**. SFN model is selected by default.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_small \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      ../
cmake --build ./ -- install
```

The following commands build Profile Small with regression test cases on **AN521** with build type **MinSizeRel**, built by **Armclang**. SFN model is selected by default.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_small \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      -DTEST_NS=ON \
      ../
cmake --build ./ -- install
```

Note:

- For devices with more constrained memory and flash requirements, it is possible to build with either only TEST_S enabled or only TEST_NS enabled. This will decrease the size of the test images. Note that both test suites must still be run to ensure correct operation.

More details of building instructions and parameters can be found TF-M build instruction guide[12].

Reference

9.4.2 Trusted Firmware-M Profile Medium-ARoT-less

Introduction

TF-M Profile Medium-ARoT-less is a reference implementation to align with security requirements defined in PSA Certified ARoT-less Level 2 protection profile (PSA Certified ARoT-less)[1].

TF-M Profile Medium-ARoT-less is defined based on TF-M Profile Medium[2], which aligns with PSA Certified Level 2 Protection Profile[3].

Overall design

TF-M Profile Medium-ARoT-less defines the following feature set:

- Firmware Framework
  - Secure Function (SFN) model[4]
  - Isolation level 1[5]
- Internal Trusted Storage (ITS)
- Crypto
  - Support both symmetric cryptography and asymmetric cryptography
  - Asymmetric key based cipher suite suggested in TLS/DTLS profiles for IoT[6] and CoAP[7], including
    - Authenticated Encryption with Associated Data (AEAD) algorithm
    - Asymmetric key algorithm based signature and verification
    - Public-key cryptography based key exchange
    - Hash function
    - HMAC for default Pseudorandom Function (PRF)
  - Asymmetric digital signature and verification for Initial Attestation Token (IAT)
- Initial Attestation
  - Asymmetric key algorithm based Initial Attestation

[12] *TF-M build instruction*

[1] SESIP Profile for PSA Certified ARoT-less Level 2

[2] *Trusted Firmware-M Profile Medium Design*

[3] SESIP Profile for PSA Certified Level 2

[4] Arm Firmware Framework for M 1.1 Extensions

[5] Arm Platform Security Architecture Firmware Framework 1.0

[6] Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things

[7] The Constrained Application Protocol (CoAP)

- Lightweight boot
  - Anti-rollback protection
  - Multiple image boot
- Firmware Update (FWU) RoT Service

Design details

Most of design in Profile Medium-ARoT-less is identical to that in Profile Medium. Refer to Profile Medium document[Page 160, 2] for details. Only the differences between Profile Medium-ARoT-less and Profile Medium are specified below.

Firmware framework

PSA Certified ARoT-less[Page 160, 1] is only applicable to devices that don't support Application RoT (ARoT) services. The platform only consists of PSA RoT domain(s) in SPE making it unnecessary to implement the isolation between ARoT and PSA RoT. Therefore, this profile selects isolation level 1 to simplify implementation and optimize memory footprint and performance.

Since only isolation level 1 is required, this profile enables SFN model rather than IPC mode for further simplification.

Protected Storage

Protected Storage (PS) is implemented as an Application RoT service in TF-M by default. Therefore, PS is disabled by default in this profile.

FWU RoT Service

As PSA Certified ARoT-less requests, FWU RoT Service is enabled by default to support secure update of platform if the platform supports FWU.

Note: Implementation note

The entire secure update sequence involves multiple agents and components, including bootloader, TF-M FWU RoT Service, image update application(s), remote server(s), etc.

The secure update sequence is implementation-defined. Here is a reference of TF-M integration with FreeRTOS OTA[8].

Implementation

The basic idea is to add dedicated profile CMake configuration files `config/profile/profile_medium_arotless` for this profile default configuration.

This top-level config file collects all the fundamental configuration flags and set them to default values, to explicitly configured the features required in this profile, during TF-M build.

A platform/use case overwrite the default settings to configure this profile.

[8] Secure OTA Updates for Cortex-M Devices with FreeRTOS

The behavior of the build flow (particularly the order of configuration loading and overriding) can be found at *Build configuration*.

The default configurations in `config/profile/profile_medium_arotless` are shown below.

Table 31:: Config flags in Profile Medium-ARoT-less top-level CMake config file

Configs	Default value	Descriptions
`TFM_ISOLATION_LEVEL`	`1`	Select level 1 isolation
`CONFIG_TFM_SPM_BACKEND`	`SFN`	Select SFN model
`TFM_PARTITION_INTERNAL_TRUSTED_STORAGE`	`ON`	Enable ITS SP
`ITS_BUF_SIZE`	`32`	ITS internal transient buffer size
`TFM_PARTITION_CRYPTO`	`ON`	Enable Crypto service
`CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED`	`OFF`	Enable Crypto asymmetric encryption operations
`TFM_MBEDCRYPTO_CONFIG_PATH`	`${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/tfm_mbedcrypto_config_profile_medium.h`	Re-use Profile Medium configuration
`TFM_MBEDCRYPTO_PSA_CRYPTO_CONFIG_PATH`	`${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/crypto_config_profile_medium.h`	Re-use Profile Medium configuration
`TFM_PARTITION_INITIAL_ATTESTATION`	`ON`	Enable Initial Attestation service
`TFM_PARTITION_FIRMWARE_UPDATE`	`ON`	Enable Firmware Update service
`TFM_PARTITION_PROTECTED_STORAGE`	`OFF`	Disable PS service
`TFM_PARTITION_PLATFORM`	`ON`	Enable TF-M Platform SP

Note: Where a configuration is the same as the default in `config/config_default.cmake`, it might be omitted from the profile configuration file.

Note: Implementation note

If the platform doesn't support secure update functionalities, FWU service will be disabled automatically. A warning will be thrown out during build.

Regression test configuration

FWU regression tests and SFN regression tests are selected by default when regression tests are enabled. Other implementations are the same as those in Profile Medium.

Platform support

To enable Profile Medium on a platform, the platform specific CMake file should be added into the platform support list in top-level Profile Medium CMake config file.

Building Profile Medium

To build Profile Medium, argument `TFM_PROFILE` in build command line should be set to `profile_medium_arotless`.

Take Musca-B1 as an example. The following commands build Profile Medium without test cases on **Musca-B1** with build type **MinSizeRel**, built by **Armclang**.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=musca-b1 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_medium_arotless \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      ../
cmake --build ./ -- install
```

The following commands build Profile Medium-ARoT-less with regression test cases on **Musca-B1** with build type **MinSizeRel**, built by **Armclang**.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=musca-b1 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_medium_arotless \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      -DTEST_S=ON -DTEST_NS=ON \
      ../
cmake --build ./ -- install
```

Note:

- For devices with more constrained memory and flash requirements, it is possible to build with either only TEST_S enabled or only TEST_NS enabled. This will decrease the size of the test images. Note that both test suites must still be run to ensure correct operation.

More details of building instructions and parameters can be found TF-M build instruction guide[9].

[9] *TF-M build instruction*

Reference

9.4.3 Trusted Firmware-M Profile Medium Design

Introduction

Compared with Profile Small, Profile Medium aims to securely connect devices to Cloud services with asymmetric cipher support. Profile Medium target devices need more resources for more cipher algorithms and higher isolation levels.

Overall design

TF-M Profile Medium defines the following feature set:

- Firmware Framework
  - Inter-Process Communication (IPC) model [PSA-FF-M]
  - Isolation level 2 [PSA-FF-M]
- Internal Trusted Storage (ITS)
- Crypto
  - Support both symmetric cryptography and asymmetric cryptography
  - Asymmetric key based cipher suite suggested in TLS/DTLS profiles for IoT [RFC7925] and CoAP [RFC7252], including
    - Authenticated Encryption with Associated Data (AEAD) algorithm
    - Asymmetric key algorithm based signature and verification
    - Public-key cryptography based key exchange
    - Hash function
    - HMAC for default Pseudorandom Function (PRF)
  - Asymmetric digital signature and verification for Initial Attestation Token (IAT)
- Initial Attestation
  - Asymmetric key algorithm based Initial Attestation
- Lightweight boot
  - Anti-rollback protection
  - Multiple image boot
- Protected Storage (PS) if off-chip storage device is integrated
  - Data confidentiality
  - Data integrity
  - Rollback protection

Design details

More details of TF-M Profile Medium design are described in following sections.

Firmware framework

Profile Medium with IPC model and isolation level 2 aims to support usage scenarios which require more complicated secure service model and additional protection to PSA RoT.

Level 2 isolation

Profile Medium selects isolation level 2 by default. In addition to isolation level 1, the PSA Root of Trust (PSA RoT) is also protected from access by the Application Root of Trust (App RoT) in level 2 isolation.

IPC model

Profile Medium enables IPC model by default. IPC model can achieve a more flexible framework and higher levels of isolation, but may require more memory footprint and bring in longer latency, compared to SFN model.

TF-M IPC model implementation follows the PSA Firmware Framework for M (PSA-FF-M) [PSA-FF-M].

Crypto service

Compared to Profile Small, Profile Medium includes asymmetric cryptography to support direct connection to Cloud services via common protocols, such as TLS/DTLS 1.2.

As suggested in CoAP [RFC7252] and [RFC7925], TF-M Profile Medium by default selects `TLS_ECDHE_ECDSA_WITH_AES_128_CCM` as reference, which requires:

- ECDHE_ECDSA as key exchange algorithm.
- AES-128-CCM (AES CCM mode with 128-bit key) as AEAD algorithm. Platforms can implement AES-128-CCM with truncated authentication tag to achieve less network bandwidth [RFC7925].
- SHA256 as Hash function.
- HMAC as Message Authentication Code algorithm.

Applications can also support TLS PSK [RFC4279] cipher suites, such as `TLS_PSK_WITH_AES_128_CCM` [RFC7925].

Note: Implementation note

Developers can replace default algorithms with others or implement more algorithms according to actual usage scenarios and device capabilities.

If a Crypto hardware accelerator is integrated, the cipher suites and algorithms also depend on those accelerator features.

More details of cipher suite are described below.

Digital signature and verification

ECDSA is selected by default in Profile Medium. ECDSA requires much shorter keys compared with RSA at the same security level. Therefore, ECDSA can cost less storage area for assets and less network bandwidth to setup a TLS connection. ECDSA is also preferred for forward compatibility of future TLS versions.

As requested in [RFC7251], ECC curve `secp256r1` should be supported. More ECC curves can be added based on the requirements in production.

If usage scenarios require RSA algorithm for backward compatibility and legacy applications, platforms can add RSA support or replace ECDSA with RSA. The cipher suite should be switched accordingly.

AEAD algorithm

If Protected Storage (PS) is implemented, it is recommended to select the same AEAD algorithm for PS service as the one used by TLS/DTLS cipher suite.

Internal Trusted Storage

The configuration of ITS is the same as those in Profile Small [PROFILE-S].

Lightweight boot

BL2 implementation can be device specific. Devices may implement diverse boot processes with different features and configurations. However, the boot loader must support anti-rollback protection. Boot loader must be able to prevent unauthorized rollback, to protect devices from being downgraded to earlier versions with known vulnerabilities.

MCUBoot in TF-M is configured as multiple image boot by default in Profile Medium. In multiple image boot, secure and non-secure images can be signed independently with different keys and they can be updated separately. It can support multiple vendors scenarios, in which non-secure and secure images are generated and updated by different vendors. Multiple image boot may require more storage area compared with single image boot.

Protected Storage

PS service is required if an off-chip storage device is integrated and used on the platform.

TF-M PS service relies on an AEAD algorithm to ensure data confidentiality and integrity. It is recommended to select the same AEAD algorithm as the one used for TLS/DTLS cipher suite.

Anti-rollback protection in PS relies on non-volatile counter(s) provided by TF-M Platform Secure Partition (SP).

Implementation

Overview

The basic idea is to add dedicated profile CMake configuration files under folder `config/profile` for TF-M Profile Medium default configuration, the same as Profile Small does.

The top-level Profile Medium config file collects all the necessary configuration flags and set them to default values, to explicitly enable the features required in Profile Medium and disable the unnecessary ones, during TF-M build.

A platform/use case can provide a configuration extension file to overwrite Profile Medium default setting and append other configurations. This configuration extension file can be added via parameter `TFM_EXTRA_CONFIG_PATH` in build command line.

The behaviour of the Profile Medium build flow (particularly the order of configuration loading and overriding) can be found at *Build configuration*

The details of configurations will be covered in each module in *Implementation details*.

Implementation details

This section discusses the details of Profile Medium implementation.

Top-level configuration files

The firmware framework configurations in `config/profile/profile_medium` are shown below.

Table 32:: Config flags in Profile Medium top-level CMake config file

Configs	Default value	Descriptions
`TFM_ISOLATION_LEVEL`	`2`	Select level 2 isolation
`TFM_PARTITION_INTERNAL_TRUSTED_STORAGE`	`ON`	Enable ITS SP
`ITS_BUF_SIZE`	`32`	ITS internal transient buffer size
`TFM_PARTITION_CRYPTO`	`ON`	Enable Crypto service
`CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED`	`OFF`	Enable Crypto asymmetric encryption operations
`TFM_MBEDCRYPTO_CONFIG_PATH`	`${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/tfm_mbedcrypto_config_profile_medium.h`	Mbed Crypto config file path
`TFM_MBEDCRYPTO_PSA_CRYPTO_CONFIG_PATH`	`${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/crypto_config_profile_medium.h`	Mbed Crypto PSA config file path
`TFM_PARTITION_INITIAL_ATTESTATION`	`ON`	Enable Initial Attestation service
`TFM_PARTITION_PROTECTED_STORAGE`	`ON`[1]	Enable PS service
`TFM_PARTITION_PLATFORM`	`ON`	Enable TF-M Platform SP

Note: Where a configuration is the same as the default in `config/config_base.cmake`, it is omitted from the profile configuration file.

[1] PS service is enabled by default. Platforms without off-chip storage devices can turn off `TFM_PARTITION_PROTECTED_STORAGE` to disable PS service. See *Protected Storage Secure Partition* for details.

Test configuration

Standard regression test configuration applies. This means that enabling regression testing via

```
-DTEST_S=ON -DTEST_NS=ON
```

Will enable testing for all enabled partitions. See above for details of enabled partitions. Because Profile Medium enables IPC model, the IPC tests are also enabled.

Some cryptography tests are disabled due to the reduced Mbed Crypto config.

Table 33:: TFM options in Profile Medium top-level CMake config file

Configs	Default value	Descriptions
TFM_CRYPTO_TEST_ALG_CBC	OFF	Disable CBC mode test
TFM_CRYPTO_TEST_ALG_CCM	ON	Enable CCM mode test
TFM_CRYPTO_TEST_ALG_CFB	OFF	Disable CFB mode test
TFM_CRYPTO_TEST_ALG_ECB	OFF	Disable ECB mode test
TFM_CRYPTO_TEST_ALG_CTR	OFF	Disable CTR mode test
TFM_CRYPTO_TEST_ALG_OFB	OFF	Disable OFB mode test
TFM_CRYPTO_TEST_ALG_GCM	OFF	Disable GCM mode test
TFM_CRYPTO_TEST_ALG_SHA_384	OFF	Disable SHA-384 algorithm test
TFM_CRYPTO_TEST_ALG_SHA_512	OFF	Disable SHA-512 algorithm test
TFM_CRYPTO_TEST_HKDF	OFF	Disable HKDF algorithm test
TFM_CRYPTO_TEST_ECDH	ON	Enable ECDH key agreement test
TFM_CRYPTO_TEST_CHACHA20	OFF	Disable ChaCha20 stream cipher test
TFM_CRYPTO_TEST_CHACHA20_POLY1305	OFF	Disable ChaCha20-Poly1305 AEAD algorithm test
TFM_CRYPTO_TEST_SINGLE_PART_FUNCS	ON	Test single-part operations in hash, MAC, AEAD and symmetric ciphers

Device configuration extension

To change default configurations and add platform specific configurations, a platform can add a platform configuration file at `platform/ext<TFM_PLATFORM>/config.cmake`

Crypto service configurations

Crypto Secure Partition

TF-M Profile Medium enables Crypto SP in top-level CMake config file. The following PSA Crypto operationts are enabled by default.

- Hash operations
- Message authentication codes
- Symmetric ciphers
- AEAD operations
- Asymmetric key algorithm based signature and verification
- Key derivation
- Key management

Mbed Crypto configurations

TF-M Profile Medium adds a dedicated Mbed Crypto config file `tfm_mbedcrypto_config_profile_medium.h` and Mbed Crypto PSA config file `crypto_config_profile_medium.h` at `/lib/ext/mbedcrypto/mbedcrypto_config` folder, instead of the common one `tfm_mbedcrypto_config_default.h` and `crypto_config_default.h` [CRYPTO-DESIGN].

Major Mbed Crypto configurations are set as listed below:

- Enable SHA256
- Enable generic message digest wrappers
- Enable AES
- Enable CCM mode for symmetric ciphers
- Disable other modes for symmetric ciphers
- Enable ECDH
- Enable ECDSA
- Select ECC curve `secp256r1`
- Other configurations required by selected option above

Other configurations can be selected to optimize the memory footprint of Crypto module.

A device/use case can append an extra config header to the Profile Medium default Mbed Crypto config file. This can be done by setting the `TFM_MBEDCRYPTO_PLATFORM_EXTRA_CONFIG_PATH` cmake variable in the platform config file `platform/ext<TFM_PLATFORM>/config.cmake`. This cmake variable is a wrapper around the `MBEDTLS_USER_CONFIG_FILE` options, but is preferred as it keeps all configuration in cmake.

Internal Trusted Storage configurations

ITS service is enabled in top-level Profile Medium CMake config file by default.

The internal transient buffer size `ITS_BUF_SIZE` [ITS-INTEGRATE] is set to 32 bytes by default. A platform/use case can overwrite the buffer size in its specific configuration extension according to its actual requirement of assets and Flash attributes.

Profile Medium CMake config file won't touch the configurations of device specific Flash hardware attributes [ITS-INTEGRATE].

Protected Storage Secure Partition

Data confidentiality, integrity and anti-rollback protection are enabled by default in PS.

If PS is selected, AES-CCM is used as AEAD algorithm by default. It requires to enable PS implementation to select diverse AEAD algorithm.

If platforms don't integrate any off-chip storage device, platforms can disable PS in platform specific configuration extension file via `platform/ext<TFM_PLATFORM>/config.cmake`.

BL2 setting

Profile Medium enables MCUBoot provided by TF-M by default. A platform can overwrite this configuration by disabling MCUBoot in its configuration extension file `platform/ext<TFM_PLATFORM>/config.cmake`.

If MCUBoot provided by TF-M is enabled, multiple image boot is selected by default in TF-M Profile Medium top-level CMake config file.

If a device implements its own boot loader, the configurations are implementation defined.

Platform support

To enable Profile Medium on a platform, the platform specific CMake file should be added into the platform support list in top-level Profile Medium CMake config file.

Building Profile Medium

To build Profile Medium, argument `TFM_PROFILE` in build command line should be set to `profile_medium`.

Take AN521 as an example:

The following commands build Profile Medium without test cases on **AN521** with build type **MinSizeRel**, built by **Armclang**.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_medium \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      ../
cmake --build ./ -- install
```

The following commands build Profile Medium with regression test cases on **AN521** with build type **MinSizeRel**, built by **Armclang**.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_medium \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      -DTEST_S=ON -DTEST_NS=ON \
      ../
cmake --build ./ -- install
```

Note:

- For devices with more contrained memory and flash requirements, it is possible to build with either only TEST_S enabled or only TEST_NS enabled. This will decrease the size of the test images. Note that both test suites must still be run to ensure correct operation.

More details of building instructions and parameters can be found TF-M build instruction guide [TFM-BUILD].

Reference

9.4.4 Trusted Firmware-M Profile Large Design

Introduction

As one of TF-M Profiles, Profile Large protects less resource-constrained Arm Cortex-M devices.

Compared to Profile Small[1] and Profile Medium[2], Profile Large aims to enable more secure features to support higher level of security required in more complex usage scenarios.

- Isolation level 3 enables additional isolation between *Application RoT* (App RoT) services.
- More crypto algorithms and cipher suites are selected to securely connect devices to remote services offered by various major Cloud Service Providers (CSP)
- Basic software countermeasures against physical attacks can be enabled.

Profile Large can be aligned as a reference implementation with the requirements defined in PSA Certified Level 3 Lightweight Protection Profile[3].

Overall design

TF-M Profile Large defines the following feature set:

- Firmware Framework
  - Inter-Process Communication (IPC) model[4]
  - Isolation level 3[Page 171, 4]
- Internal Trusted Storage (ITS)
- Crypto
  - Support both symmetric ciphers and asymmetric ciphers
  - Asymmetric key based cipher suites defined in TLS 1.2[5] to support direct secure connection to major CSPs, including
    - Authenticated Encryption with Associated Data (AEAD) algorithm
    - Asymmetric key algorithm based signature and verification
    - Public-key cryptography based key exchange
    - Hash function
    - HMAC for default Pseudorandom Function (PRF)
  - Asymmetric digital signature and verification for Initial Attestation Token (IAT)
  - Asymmetric algorithms for firmware image signature verification

[1] *Trusted Firmware-M Profile Small Design*
[2] *Trusted Firmware-M Profile Medium Design*
[3] PSA Certified Level 3 Lightweight Protection Profile
[4] Arm Platform Security Architecture Firmware Framework 1.0
[5] The Transport Layer Security (TLS) Protocol Version 1.2

  - Key derivation
- Initial Attestation
  - Asymmetric key algorithm based Initial Attestation
- **Secure boot**
    - Anti-rollback protection
    - Multiple image boot
- Protected Storage (PS) if off-chip storage device is integrated
  - Data confidentiality
  - Data integrity
  - Rollback protection
- Software countermeasures against physical attacks

Design details

More details of TF-M Profile Large design are described in following sections.

Firmware framework

Profile Large selects IPC model and isolation level 3 by default.

Isolation level 3 supports additional isolation between App RoT services, compared to isolation level 2. It can protect *RoT* services from each other when their vendors don't trust each other.

Crypto service

Profile Large supports direct connection to Cloud services via common protocols, such as TLS 1.2.

In some usage scenarios, PSA RoT can be managed by device manufacturer or other vendors and is out of control of application developers. Profile Large selects alternative crypto algorithms for each crypto function to support multiple common cipher suites required by various major CSPs. Therefore, application developers can support services for diverse CSPs on same devices with Profile Large, without relying on PSA RoT upgrades of crypto.

Devices meeting Profile Large should be in a position to offer at least two alternatives to every cryptographic primitive for symmetric, asymmetric and hash, and be able to use them for encryption, AEAD, signature and verification.

It will cost more resource in Profile Large to support more crypto algorithms and cipher suites, compared to Profile Medium[2].

Boot loader

BL2 implementation can be device specific. Devices may implement diverse boot processes with different features and configurations. However, the boot loader must support anti-rollback protection. Boot loader must be able to prevent unauthorized rollback, to protect devices from being downgraded to earlier versions with known vulnerabilities.

MCUBoot in TF-M is configured as multiple image boot by default in Profile Large. In multiple image boot, secure and non-secure images can be signed independently with different keys and they can be updated separately. It can support multiple vendors scenarios, in which non-secure and secure images are generated and updated by different vendors. Multiple image boot may cost larger memory footprint compared with single image boot.

Boot loader can implement software countermeasures to mitigate physical attacks.

Protected Storage

PS service is required if an off-chip storage device is integrated and used on the platform.

Anti-rollback protection in PS relies on non-volatile counter(s) provided by TF-M Platform *Secure Partition* (SP).

Software countermeasures against physical attacks

TF-M Profile Large enables TF-M Fault Injection Hardening (FIH) library Profile Medium by default. It enables the following countermeasure techniques:

- Control flow monitor
- Failure loop hardening
- Complex constants
- Redundant variables and condition checks

Refer to TF-M physical attack mitigation design document[6] for FIH library details.

Note: TF-M FIH library is still under development.

TF-M FIH library hardens TF-M critical execution steps to make physical attacks more difficult, together with device hardware countermeasures. It is not guaranteed that TF-M FIH library is able to mitigate all kinds of physical attacks.

Note: Implementation note

TF-M FIH library doesn't cover platform specific critical configurations. Platforms shall implement software countermeasures against physical attacks to protect platform specific implementation.

[6] *Physical attack mitigation in Trusted Firmware-M*

Implementation

Overview

The basic idea is to add dedicated profile CMake configuration files under folder `config/profile` for TF-M Profile Large default configuration, the same as other TF-M Profiles do.

The top-level Profile Large config file collects all the necessary configuration flags and set them to default values, to explicitly enable the features required in Profile Large and disable the unnecessary ones, during TF-M build.

A platform/use case can provide a configuration extension file to overwrite Profile Large default setting and append other configurations. This configuration extension file can be added via parameter `TFM_EXTRA_CONFIG_PATH` in build command line.

The behaviour of the Profile Large build flow (particularly the order of configuration loading and overriding) can be found at *Build configuration*

The details of configurations will be covered in each module in *Implementation details*.

Implementation details

This section discusses the details of Profile Large implementation.

Top-level configuration files

The firmware framework configurations in `config/profile/profile_large` are shown below.

Table 34:: Config flags in Profile Large top-level CMake config file

Configs	Descriptions	Default value
`TFM_ISOLATION_LEVEL`	Select level 3 isolation	`3`
`TFM_PARTITION_INTERNAL_TRUSTED_STORAGE`	Enable ITS SP	`ON`
`TFM_PARTITION_CRYPTO`	Enable Crypto service	`ON`
`TFM_MBEDCRYPTO_CONFIG_PATH`	MbedTLS config file path	`${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/tfm_mbedcrypto_config_profile_large.h`
`TFM_MBEDCRYPTO_PSA_CRYPTO_CONFIG_PATH`	MbedTLS PSA config file path	`${CMAKE_SOURCE_DIR}/lib/ext/mbedcrypto/mbedcrypto_config/crypto_config_profile_large.h`
`TFM_PARTITION_INITIAL_ATTESTATION`	Enable Initial Attestation service	`ON`
`TFM_PARTITION_PROTECTED_STORAGE` [a]	Enable PS service	`ON`
`TFM_PARTITION_PLATFORM`	Enable TF-M Platform SP	`ON`

Crypto service configurations

Crypto Secure Partition

TF-M Profile Large enables Crypto SP in top-level CMake config file and selects all the Crypto modules.

MbedTLS configurations

TF-M Profile Large adds a dedicated MbedTLS config file `tfm_mbedcrypto_config_profile_large.h` and MbedTLS PSA config file `crypto_config_profile_large.h` under `/lib/ext/mbedcrypto/mbedcrypto_config` folder, instead of the common one `tfm_mbedcrypto_config_default.h` and `crypto_config_default.h`[7].

Major MbedTLS configurations are set as listed below:

- Enable SHA256, SHA384 and SHA512
- Enable generic message digest wrappers
- Enable AES
- Enable CCM mode, GCM mode, CTR mode, CFB mode and CBC mode for symmetric ciphers
- Disable other modes for symmetric ciphers
- Enable ECDH
- Enable ECDSA
- Enable RSA
- Select ECC curve `secp256r1` and `secp384r1`
- Enable HMAC-based key derivation function
- Other configurations required by selected option above

A device/use case can append an extra config header to the Profile Large default MbedTLS config file to override the default settings. This can be done by setting the `TFM_MBEDCRYPTO_PLATFORM_EXTRA_CONFIG_PATH` cmake variable in the platform config file `platform/ext<TFM_PLATFORM>/config.cmake`. This cmake variable is a wrapper around the `MBEDTLS_USER_CONFIG_FILE` options, but is preferred as it keeps all configuration in cmake.

Internal Trusted Storage configurations

ITS service is enabled in top-level Profile Large CMake config file by default.

The internal transient buffer size `ITS_BUF_SIZE`[8] is set to 64 bytes by default. A platform/use case can overwrite the buffer size in its specific configuration extension according to its actual requirement of assets and Flash attributes.

Profile Large CMake config file won't touch the configurations of device specific Flash hardware attributes.

[7] *Crypto design*

[8] *ITS integration guide*

Protected Storage Secure Partition

Data confidentiality, integrity and anti-rollback protection are enabled by default in PS.

If PS is selected, AES-CCM is used as AEAD algorithm by default. If platform hardware crypto accelerator supports the AEAD algorithm, the AEAD operations can be executed in hardware crypto accelerator.

If platforms don't integrate any off-chip storage device, platforms can disable PS in platform specific configuration extension file via `platform/ext<TFM_PLATFORM>/config.cmake`.

BL2 setting

Profile Large enables MCUBoot provided by TF-M by default. A platform can overwrite this configuration by disabling MCUBoot in its configuration extension file `platform/ext<TFM_PLATFORM>/config.cmake`.

If MCUBoot provided by TF-M is enabled, multiple image boot is selected by default.

If a device implements its own boot loader, the configurations are implementation defined.

Software countermeasure against physical attacks

Profile Large selects TF-M FIH library Profile Medium by specifying `-DTFM_FIH_PROFILE=MEDIUM` in top-level CMake config file.

System integrators shall implement software countermeasures in platform specific implementations.

Device configuration extension

To change default configurations and add platform specific configurations, a platform can add a platform configuration file at `platform/ext<TFM_PLATFORM>/config.cmake`

Test configuration

Some cryptography tests are disabled due to the reduced MbedTLS config. Profile Large specific test configurations are also specified in Profile Large top-level CMake config file `config/profile/profile_large_test.cmake`.

Table 35:: Profile Large crypto test configuration

Configs	Default value	Descriptions
`TFM_CRYPTO_TEST_ALG_CBC`	`ON`	Test CBC cryptography mode
`TFM_CRYPTO_TEST_ALG_CCM`	`ON`	Test CCM cryptography mode
`TFM_CRYPTO_TEST_ALG_CFB`	`OFF`	Test CFB cryptography mode
`TFM_CRYPTO_TEST_ALG_ECB`	`OFF`	Test ECB cryptography mode
`TFM_CRYPTO_TEST_ALG_CTR`	`OFF`	Test CTR cryptography mode
`TFM_CRYPTO_TEST_ALG_OFB`	`OFF`	Test OFB cryptography mode
`TFM_CRYPTO_TEST_ALG_GCM`	`ON`	Test GCM cryptography mode
`TFM_CRYPTO_TEST_ALG_SHA_384`	`OFF`	Test SHA-384 cryptography algorithm
`TFM_CRYPTO_TEST_ALG_SHA_512`	`ON`	Test SHA-512 cryptography algorithm
`TFM_CRYPTO_TEST_HKDF`	`ON`	Test HMAC-based key derivation function
`TFM_CRYPTO_TEST_ECDH`	`ON`	Test ECDH key agreement algorithm
`TFM_CRYPTO_TEST_CHACHA20`	`OFF`	Test ChaCha20 stream cipher
`TFM_CRYPTO_TEST_CHACHA20_POLY1305`	`OFF`	Test ChaCha20-Poly1305 AEAD algorithm
`TFM_CRYPTO_TEST_SINGLE_PART_FUNCS`	`OFF`	Test single-part operations in hash, MAC, AEAD and symmetric ciphers

Platform support

To enable Profile Large on a platform, the platform specific CMake file should be added into the platform support list in top-level Profile Large CMake config file.

Building Profile Large

To build Profile Large, argument `TFM_PROFILE` in build command line should be set to `profile_large`.

Take AN521 as an example:

The following commands build Profile Large without test cases on **AN521** with build type **MinSizeRel**, built by **Armclang**.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_large \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      ../
cmake --build ./ -- install
```

The following commands build Profile Large with regression test cases on **AN521** with build type **MinSizeRel**, built by **Armclang**.

```
cd <TFM root dir>
mkdir build && cd build
cmake -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_TOOLCHAIN_FILE=../toolchain_ARMCLANG.cmake \
      -DTFM_PROFILE=profile_large \
      -DCMAKE_BUILD_TYPE=MinSizeRel \
      -DTEST_S=ON -DTEST_NS=ON \
      ../
cmake --build ./ -- install
```

More details of building instructions and parameters can be found TF-M build instruction guide[9].

Reference

Option	Base	Small	ARoT-less	Medium	Large
TFM_ISOLATION_LEVEL	1	1	1	2	3
CONFIG_TFM_SPM_BACKEND	SFN	SFN	SFN	IPC	IPC
TFM_PARTITION_CRYPTO	OFF	ON	ON	ON	ON
TFM_PARTITION_INTERNAL_TRUSTED_STORAGE	OFF	ON	ON	ON	ON
TFM_PARTITION_PLATFORM	OFF	OFF	ON	ON	ON
TFM_PARTITION_PROTECTED_STORAGE	OFF	OFF	OFF	ON	ON
TFM_PARTITION_INITIAL_ATTESTATION	OFF	ON	ON	ON	ON
SYMMETRIC_INITIAL_ATTESTATION	OFF	ON	OFF	OFF	OFF
TFM_PARTITION_FIRMWARE_UPDATE	OFF	OFF	ON	OFF	OFF
PS_CRYPTO_AEAD_ALG	GCM	-	-	CCM	CCM
PSA_FRAMEWORK_HAS_MM_IOVEC	OFF	ON	OFF	OFF	OFF
MCUBOOT_IMAGE_NUMBER [1]	2	1	2	2	2
Advanced options, defined in the corresponded header (.h) file					
CRYPTO_ENGINE_BUF_SIZE	0x2080	0x400	0x2080	0x2080	0x2380
CRYPTO_ASYM_SIGN_MODULE_ENABLED	ON	OFF	ON	ON	ON
CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED	ON	OFF	OFF	OFF	ON
CRYPTO_SINGLE_PART_FUNCS_DISABLED	OFF	ON	OFF	OFF	OFF
CRYPTO_CONC_OPER_NUM	8	4	8	8	8
CONFIG_TFM_CONN_HANDLE_MAX_NUM	8	3	8	8	8
ITS_BUF_SIZE [2]	512	32	32	32	512

1. *MCUBOOT_IMAGGE_NUMBER* value is taken from MCUBoot default configuration, execept profile Small.
2. Many platforms redefine *ITS_BUF_SIZE* value.

Each profile has predefined configuration for cryptographic library, located in `/lib/ext/mbedcrypto/mbedcrypto_config/`

---

[9] *TF-M build instruction*

9.5 Tests configuration

9.5.1 Regression tests

Regression test configuration is controlled entirely by TEST_NS or TEST_S or single test suite configuration. The group test configurations and single test suite configurations are listed below, all of them are disabled by default.

Parameter	Description
TEST_NS_ATTESTATION	Build non-secure regression Attestation tests.
TEST_NS_T_COSE	Build non-secure regression t_cose tests.
TEST_NS_QCBOR	Build non-secure regression QCBOR tests.
TEST_NS_CRYPTO	Build non-secure regression Crypto tests.
TEST_NS_ITS	Build non-secure regression ITS tests.
TEST_NS_PS	Build non-secure regression PS tests.
TEST_NS_PLATFORM	Build non-secure regression Platform tests.
TEST_NS_FWU	Build non-secure regression FWU tests.
TEST_NS_IPC	Build non-secure regression IPC tests.
TEST_NS_SLIH_IRQ	Build non-secure regression Second-Level Interrupt Handling tests.
TEST_NS_FLIH_IRQ	Build non-secure regression First-Level Interrupt Handling tests.
TEST_NS_MULTI_CORE	Build non-secure regression multi-core tests.
TEST_S_ATTESTATION	Build secure regression Attestation tests.
TEST_S_CRYPTO	Build secure regression Crypto tests.
TEST_S_ITS	Build secure regression ITS tests.
TEST_S_PS	Build secure regression PS tests.
TEST_S_PLATFORM	Build secure regression Platform tests.
TEST_S_FWU	Build secure regression FWU tests.
TEST_S_IPC	Build secure regression IPC tests.

Individual test suites can be enabled when their dependencies like partitions or other specific configurations are set. On the one hand, some test suites depend on other test suites. On the other hand, some test suites conflict with other test suites. Test configurations and dependencies will be checked in ${TFM_TEST_REPO_PATH}/test/config/check_config.cmake.

If regression testing is enabled by TEST_NS or TEST_S, individual test suites will be enabled or disabled as appropriate for the TF-M configuration (i.e. all enabled secure partitions will be tested).

Some cryptographic tests can be enabled and disabled. This is done to prevent false failures from being reported when a smaller Mbed Crypto config is being used which does not support all features.

Parameter	Description	Default value
TFM_CRYPTO_TEST_ALG_CBC	Test CBC cryptography mode	ON
TFM_CRYPTO_TEST_ALG_CCM	Test CCM cryptography mode	ON
TFM_CRYPTO_TEST_ALG_CFB	Test CFB cryptography mode	ON
TFM_CRYPTO_TEST_ALG_ECB	Test ECB cryptography mode	ON
TFM_CRYPTO_TEST_ALG_CTR	Test CTR cryptography mode	ON
TFM_CRYPTO_TEST_ALG_OFB	Test OFB cryptography mode	ON
TFM_CRYPTO_TEST_ALG_GCM	Test GCM cryptography mode	ON
TFM_CRYPTO_TEST_ALG_SHA_384	Test SHA-384 cryptography algorithm	ON
TFM_CRYPTO_TEST_ALG_SHA_512	Test SHA-512 cryptography algorithm	ON
TFM_CRYPTO_TEST_HKDF	Test HKDF key derivation algorithm	ON
TFM_CRYPTO_TEST_ECDH	Test ECDH key agreement algorithm	ON
TFM_CRYPTO_TEST_CHACHA20	Test ChaCha20 stream cipher	ON
TFM_CRYPTO_TEST_ALG_CHACHA20_POLY1305	Test ChaCha20-Poly1305 AEAD algorithm	ON

9.5.2 PSA tests

PSA tests are configured by using the `TEST_PSA_API` cmake variable. The variable should be set to the name of the test suite that is desired. It is _not_ supported to set both `TEST_PSA_API` and `TEST_S` or `TEST_NS`.

The Functional API tests are:

- `CRYPTO`
- `INITIAL_ATTESTATION`
- `STORAGE` (INTERNAL_TRUSTED_STORAGE and PROTECTED_STORAGE)
- `INTERNAL_TRUSTED_STORAGE`
- `PROTECTED_STORAGE`

The Firmware Framework test suites are:

- `IPC`

Note that these map directly to the `SUITE` cmake variable used in the psa-arch-tests documentation.

TF-M is highly configurable project with many configuration options to meet a user's needs. A user can select the desired set of services and fine-tune them to their requirements. There are two types of configuration options

Build configuration

Specifies which file or component to include into compilation and build. These are options, usually used by a build system to enable/disable modules, specify location of external dependency or other selection, global to a project. These option set shall be considered while adopting TF-M to other build systems. In the *Base Configuration* table these options have *Build* type.

Component configuration

To adjust a particular parameter to a desired value. Those options are local to a component or externally referenced when components are coupled. Options are in C header file. The *The Header File Config System* has more details about it. In the *Base Configuration* table these options have *Component* type.

Note: Originally, TF-M used CMake variables for both building and component tuning purposes. It was convenient to have a single system for both building and component's configurations. To simplify and improve configurability and better support build systems other than a CMake, TF-M introduced a *The Header File Config System* and moved component options into a dedicated config headers.

9.6 How to configure

TF-M Project provides a base build, defined in `/config/config_base.cmake` and `/config/config_base.h`. Starting from the base, users can enable required services and features using several independent methods to configure TF-M.

Use *TF-M Profiles*.
There are 4 sets of predefined configurations for a elected use cases, called profiles. A user can select a profile by providing -DTFM_PROFILE=<profile file name>. Each profiles represented by a pair of configuration files for Building (CMake) options and Component options (.h file)

Use a custom profile.
Another method is to take an existing TF-M profile and adjust the desired options manually editing CMake and config header files. This is for users familiar with TF-M.

Use *The Kconfig System*.
This method is recommended for beginners. Starting from the *base configuration* a user can enable necessary services and options. KConfig ensurers that all selected options are consistent and valid. This is new in v1.7.0 and it covers only SPM and PSA services. As an output KConfig produces a pair of configuration files, similar to a profile.

Note: In contrast, before TF-M v1.7.0, the default build includes all possible features. With growing functionality, such rich default build became unpractical by not fitting into every platform and confusing of big memory requirements.

9.7 Priorities

A project configuration performed in multiple steps with priorities. The list below explains the process but for the details specific to *Build configuration* or *The Header File Config System* please check the corresponded document.

1. The base configuration with default values is used as a starting point
2. A profile options applied on top of the base
3. A platform can check the selected configuration and apply restrictions
4. Finally, command line options can modify the composed set

Note: To ensure a clear intention and conscious choice, all options must be provided explicitly via a project configuration file. Default values on step 1 will generate warnings which are expected to break a build.

9.8 Base Configuration

The base configuration is the ground for configuring TF-M, provided defaults are defined in `/config/config_base.cmake` and `/config/config_base.h`. The base build includes SPM and platform code only.

This table lists the config option categorizations of the SPM and Secure Partitions.

9.8.1 Crypto

Options	Type	Base Value
TFM_PARTITION_CRYPTO	Build	OFF
CRYPTO_TFM_BUILTIN_KEYS_DRIVER	Build	ON
CRYPTO_NV_SEED	Component	ON
CRYPTO_ENGINE_BUF_SIZE	Component	0x2080
CRYPTO_IOVEC_BUFFER_SIZE	Component	5120
CRYPTO_STACK_SIZE	Component	0x1B00
CRYPTO_CONC_OPER_NUM	Component	8
CRYPTO_RNG_MODULE_ENABLED	Component	1
CRYPTO_KEY_MODULE_ENABLED	Component	1
CRYPTO_AEAD_MODULE_ENABLED	Component	1
CRYPTO_MAC_MODULE_ENABLED	Component	1
CRYPTO_HASH_MODULE_ENABLED	Component	1
CRYPTO_CIPHER_MODULE_ENABLED	Component	1
CRYPTO_ASYM_SIGN_MODULE_ENABLED	Component	1
CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED	Component	1
CRYPTO_KEY_DERIVATION_MODULE_ENABLED	Component	1
CRYPTO_SINGLE_PART_FUNCS_ENABLED	Component	1

9.8.2 Initial Attestation

Options	Type	Base Value
TFM_PARTITION_INITIAL_ATTESTATION	Build	OFF
SYMMETRIC_INITIAL_ATTESTATION	Build	OFF
ATTEST_INCLUDE_TEST_CODE	Build	OFF
ATTEST_KEY_BITS	Build	256
ATTEST_TOKEN_PROFILE	Component	“PSA_IOT_1”
ATTEST_INCLUDE_OPTIONAL_CLAIMS	Component	1
ATTEST_INCLUDE_COSE_KEY_ID	Component	0
ATTEST_STACK_SIZE	Component	0x700

9.8.3 Internal Trusted Storage

Options	Type	Base Value
TFM_PARTITION_INTERNAL_TRUSTED_STORAGE	Build	OFF
ITS_CREATE_FLASH_LAYOUT	Component	1
ITS_RAM_FS	Component	0
ITS_VALIDATE_METADATA_FROM_FLASH	Component	1
ITS_MAX_ASSET_SIZE	Component	512
ITS_NUM_ASSETS	Component	10
ITS_BUF_SIZE	Component	ITS_MAX_ASSET_SIZE
ITS_STACK_SIZE	Component	0x720

9.8.4 Protected Storage

Options	Type	Base Value
TFM_PARTITION_PROTECTED_STORAGE	Build	OFF
PS_ENCRYPTION	Build	ON
PS_CRYPTO_AEAD_ALG	Build	PSA_ALG_GCM
PS_CREATE_FLASH_LAYOUT	Component	1
PS_RAM_FS	Component	0
PS_VALIDATE_METADATA_FROM_FLASH	Component	1
PS_MAX_ASSET_SIZE	Component	2048
PS_NUM_ASSETS	Component	10
PS_ROLLBACK_PROTECTION	Component	1
PS_STACK_SIZE	Component	0x700

9.8.5 Firmware Update

Options	Type	Base Value
PLATFORM_HAS_FIRMWARE_UPDATE_SUPPORT	Build	OFF
TFM_PARTITION_FIRMWARE_UPDATE	Build	OFF
TFM_CONFIG_FWU_MAX_WRITE_SIZE	Build	1024
TFM_CONFIG_FWU_MAX_MANIFEST_SIZE	Build	0
FWU_DEVICE_CONFIG_FILE	Build	“”
FWU_SUPPORT_TRIAL_STATE	Build	Depends on MCUBOOT_UPGRADE_STRATEGY
TFM_FWU_BOOTLOADER_LIB	Build	“mcuboot”
TFM_FWU_BUF_SIZE	Component	PSA_FWU_MAX_BLOCK_SIZE
FWU_STACK_SIZE	Component	0x600

9.8.6 Platform Secure Partition

Options	Type	Base Value
TFM_PARTITION_PLATFORM	Build	OFF
PLATFORM_SERVICE_INPUT_BUFFER_SIZE	Component	64
PLATFORM_SERVICE_OUTPUT_BUFFER_SIZE	Component	64
PLATFORM_SP_STACK_SIZE	Component	0x500
PLATFORM_NV_COUNTER_MODULE_DISABLED	Component	0

9.8.7 NS Agent Mailbox Secure Partition

Options	Type	Base Value
NS_AGENT_MAILBOX_STACK_SIZE	Component	0x800

9.8.8 Secure Partition Manager

Options	Type	Base Values
TFM_ISOLATION_LEVEL	Build	1
PSA_FRAMEWORK_HAS_MM_IOVEC	Build	OFF
CONFIG_TFM_SPM_BACKEND	Build	“SFN”
TFM_SPM_LOG_LEVEL	Build	1
CONFIG_TFM_CONN_HANDLE_MAX_NUM	Component	8
CONFIG_TFM_DOORBELL_API	Component	0
CONFIG_TFM_SCHEDULE_WHEN_NS_INTERRUPTED	Component	0

CHAPTER

TEN

INTEGRATION GUIDE

The purpose of this document is to provide a guide on how to integrate TF-M with other hardware platforms and operating systems.

10.1 Source Structure

TF-M is designed to provide a user-friendly source structure to ease integration and service development. This document introduces the source structure and its design intention of TF-M. Additionally, the folders below are important for TF-M integration and descibed in a separate documents:

10.1.1 Details for the platform folder

Interfacing with TF-M

platform/ext/target/tfm_peripherals_def.h

This file should enumerate the hardware peripherals that are available for TF-M on the platform. The name of the peripheral used by a service is set in its manifest file. The platform have to provide a macro for each of the provided peripherals, that is substituted to a pointer to type `struct platform_data_t`. The memory that the pointer points to is allocated by the platform code. The pointer gets stored in the partitions database in SPM, and it is provided to the SPM HAL functions.

Peripherals currently used by the services in TF-M

- `TFM_PERIPHERAL_UART1`- UART1

Note: If a platform doesn't support a peripheral, that is used by a service, then the service cannot be used on the given platform. Using a peripheral in TF-M that is not supported by the platform results in compile error.

platform/include/tfm_spm_hal.h

This file contains the declarations of functions that a platform implementation has to provide for TF-M's SPM. For details see the comments in the file.

platform/include/tfm_platform_system.h

This file contains the declarations of functions that a platform implementation has to provide for TF-M's Platform Service. For details see `docs/user_guides/services/tfm_platform_integration_guide.rst`

System Startup

Before calling `main()`, platform startup code should initialise all system clocks, perform runtime initialisation and other steps such as setting the vector table. Configuration of the following features is optional as TF-M architecture code will check and initialise them:

- The Security Extension.
- The Floating-point Extension.

Debug authentication settings

A platform may provide the option to configure debug authentication. TF-M core calls the HAL function `enum tfm_hal_status_t tfm_hal_platform_init(void)` in which debug authentication is configured based on the following defines:

- *DAUTH_NONE*: Debugging the system is not enabled.
- *DAUTH_NS_ONLY*: Invasive and non invasive debugging of non-secure code is enabled.
- *DAUTH_FULL*: Invasive and non-invasive debugging of non-secure and secure code is enabled.
- *DAUTH_CHIP_DEFAULT*: The debug authentication options are used that are set by the chip vendor.

The desired debug authentication configuration can be selected by setting one of the options above to the cmake command with the *-DDEBUG_AUTHENTICATION="<define>"* option. The default value of *DEBUG_AUTHENTICATION* is *DAUTH_CHIP_DEFAULT*.

Note: `enum tfm_hal_status_t tfm_hal_platform_init(void)` is called during the TF-M core initialisation phase, before initialising secure partition. This means that BL2 runs with the chip default setting.

Sub-folders

include

This folder contains the interfaces that TF-M expects every target to provide. The code in this folder is created as a part of the TF-M project therefore it adheres to the BSD 3.0 license.

ext

This folder contains code that has been imported from other projects so it may have licenses other than the BSD 3.0 used by the TF-M project.

Please see the *Details for the platform/ext folder* for details.

10.1.2 Details for the platform/ext folder

This folder has code that has been imported from other projects. This means the files in this folder and subfolders have Apache 2.0 license which is different to BSD 3.0 license applied to the parent TF-M project.

Note: This folder is strictly Apache 2.0 with the exception of cmake files. Maintainers should be consulted if this needs to be revisited.

Sub-folders

accelerator

This folder contains cmake and code files to interact cryptographic accelerators.

In order to use a cryptographic accelerator, a platform must set `CRYPTO_HW_ACCELERATOR_TYPE` in preload.cmake. This option maps directly to the subdirectories of the accelerator directory. Currently available accelerators are : the CryptoCell `cc312`, the STMicroelectronics accelerator `stm` .

A minimal API is exposed to interact with accelerators, the details of this api are in `accelerator/interface/crypto_hw.h`. Implementation of the API is inside the subdirectory of the individual accelerator.

To configure a cryptographic accelerator at build time, two cmake options can be specified.

- **`CRYPTO_HW_ACCELERATOR`**
  - `ON` All possible mbedtls cryptographic operations will be offloaded to the accelerator.
  - `OFF` The cryptographic accelerator will be ignored and software cryptography will be used.

cmsis

This folder contains core and compiler specific header files imported from the `CMSIS_5` project.

common

armclang and gcc

These contain the linker scripts used to configure the memory regions in TF-M regions.

template

This directory contains platform-independent dummy implementations of the interfaces in `platform/include`. These implementations can be built directly for initial testing of a platform port, or used as a basic template for a real implementation for a particular target. They **must not** be used in production systems.

driver

This folder contains the headers with CMSIS compliant driver definitions that that TF-M project expects a target to provide.

target_cfg.h

This file is expected to define the following macros respectively.

- `TFM_DRIVER_STDIO` - This macro should expand to a structure of type `ARM_DRIVER_USART`. TFM redirects its standard input and output to this instance of USART.
- `NS_DRIVER_STDIO` - This macro should expand to a structure of type `ARM_DRIVER_USART`. Non-Secure application redirects its standard input and output to this instance of USART.

target

This folder contains the files for individual target. For a buildable target, the directory path from the `target` directory to its `CMakeLists.txt` file is the argument that would be given to `-DTFM_PLATFORM=`.

The standard directory structure is as follows:

- **target**
  - **<Vendor name>**
    - common
    - <buildable target 1>
    - <buildable target 2>
    - <buildable target 3>

Each buildable target must contain the cmake files mandated in the section below.

The `common` directory is not required, but can be used to contain code that is used by multiple targets.

There must not be any directories inside the vendor directory that is not either the `common` directory or a buildable platform, to avoid confusion about what directories are a valid `TFM_PLATFORM`.

Buildable target required cmake files

A buildable target must provide 3 mandatory cmake files. These files must all be placed in the root of the buildable target directory.

preload.cmake

This file contains variable definitions that relate to the underlying hardware of the target.

- TFM_SYSTEM_PROCESSOR: The processor used by the target. The format is that same as the format used in the -mcpu= argument of GNUARM or ARMCLANG. The special +modifier syntax must not be used.
- TFM_SYSTEM_ARCHITECTURE: The architecture used by the target. The format is that same as the format used in the -march= argument of GNUARM or ARMCLANG. The special +modifier syntax must not be used.
- TFM_SYSTEM_DSP: Whether the target has the DSP feature of the given TFM_SYSTEM_PROCESSOR
- CRYPTO_HW_ACCELERATOR_TYPE: The type of cryptographic accelerator the target has, if it has one. This maps exactly to the subdirectories of platform/ext/accelerator
- CONFIG_TFM_FP_ARCH: The "FPU architecture" (Armclang) or "FP hardware" (GNU) used by the target for compiling C source files. The value will be transferred to -mfpu argument of GNUARM or ARMCLANG
- CONFIG_TFM_FP_ARCH_ASM: The "FPU architecture" (Armclang) used by the target for assembling ASM source files. This value will be transferred to the --fpu argument of ARMCLANG. This value is not used by GNUARM.

For more details on CONFIG_TFM_FP_ARCH and CONFIG_TFM_FP_ARCH_ASM, please refer to *Floating-Point Support*.

Other than these particular cmake variables, it is permissible for the preload.cmake file to contain add_definitions statements, in order for set compile definitions that are global for the hardware. This is commonly used to select a particular set of code from a vendor SDK.

It is not permissible to contains code other than the above in a preload.cmake file, any general cmake code should be placed in CMakeLists.txt and any configuration options should be contained in config.cmake

config.cmake

This file collects platform-specific overrides to the configuration options. This should only contain cmake options that are included in config_base.cmake. These options should be set as CACHE variables, as they are in config_base.cmake.

CMakeLists.txt

This file should contain all other required cmake code for the platform. This primarily consists of the following:

- Adding an include directory to the target platform_region_defs, which contains the headers flash_layout.h and region_defs.h
- Adding startup and scatter files to the tfm_s, tfm_ns and bl2 targets.
- linking CMSIS_5_tfm_ns to the correct version of the CMSIS RTX libraries, as defined in lib/ext/CMSIS_5/CMakeLists.txt
- Adding required source files, include directories and compile definitions to the platform_s, platform_ns and platform_bl2 targets.

preload_ns.cmake

This optional cmake file is required only if the target runs the NSPE on a core that requires different compiler options than the SPE core. This file has the same format as `preload.cmake`, but instead details the hardware of the NS core that is **not** running the main TF-M secure code.

install.cmake

This optional cmake file is required only if additional files need to be installed for the platform.

Flash layout header file

Target must provide a header file, called `flash_layout.h`, which defines the information explained in the follow subsections. The defines must be named as they are in the subsections.

BL2 bootloader

The BL2 bootloader requires the following definitions:

- `FLASH_BASE_ADDRESS` - Defines the first valid address in the flash.
- `FLASH_AREA_BL2_OFFSET` - Defines the offset from the flash base address where the BL2 - MCUBOOT area starts.
- `FLASH_AREA_BL2_SIZE` - Defines the size of the BL2 area.
- `FLASH_AREA_SCRATCH_OFFSET` - Defines the offset from the flash base address where the scratch area starts, which is used during image swapping.
- `FLASH_AREA_SCRATCH_SIZE` - Defines the size of the scratch area. The minimal size must be as the biggest sector size in the flash.
- `FLASH_DEV_NAME` - Specifies the flash device used by BL2.

The BL2 requires further definitions depending on the number of images, the meaning of these macros are also slightly different:

- Required definitions in case of 1 image (S and NS images are concatenated and handled together as one binary blob):
  - `FLASH_AREA_0_OFFSET` - Defines the offset from the flash base address where the primary image area starts, which hosts the active firmware image.
  - `FLASH_AREA_0_SIZE` - Defines the size of the primary image area.
  - `FLASH_AREA_2_OFFSET` - Defines the offset from the flash base address where the secondary image area starts, which is a placeholder for new firmware images.
  - `FLASH_AREA_2_SIZE` - Defines the size of the secondary image area.
- Required definitions in case of 2 images (S and NS images are handled and updated separately):
  - `FLASH_AREA_0_OFFSET` - Defines the offset from the flash base address where the primary image areas start, which host the active firmware images. It is also the offset of the primary (active) secure image area.
  - `FLASH_AREA_0_SIZE` - Defines the size of the primary secure image area.
  - `FLASH_AREA_1_OFFSET` - Defines the offset from the flash base address where the primary (active) non-secure image area starts.

  - FLASH_AREA_1_SIZE - Defines the size of the primary non-secure image area.
  - FLASH_AREA_2_OFFSET - Defines the offset from the flash base address where the secondary image areas start, which are placeholders for new firmware images. It is also the offset of the secondary secure image area.
  - FLASH_AREA_2_SIZE - Defines the size of the secondary secure image area.
  - FLASH_AREA_3_OFFSET - Defines the offset from the flash base address where the secondary non-secure image area starts.
  - FLASH_AREA_3_SIZE - Defines the size of the secondary non-secure image area.

The table below shows a fraction of the flash layout in case of 2 and 1 updatable images with the related flash areas that hold the firmware images:

Image number: 2		Image number: 1	
Flash area	**Content**	**Flash area**	**Content**
FLASH_AREA_0	Secure image primary slot	FLASH_AREA_0	Secure + Non-secure image primary slot
FLASH_AREA_1	Non-secure image primary slot		
FLASH_AREA_2	Secure image secondary slot	FLASH_AREA_2	Secure + Non-secure image secondary slot
FLASH_AREA_3	Non-secure image secondary slot		
FLASH_AREA_SCRATCH	Scratch area	FLASH_AREA_SCRATCH	Scratch area

- IMAGE_EXECUTABLE_RAM_START - Defines the start of the region to which images are allowed to be loaded. Only used if MCUBOOT_UPGRADE_STRATEGY is configured to be RAM_LOAD.
- IMAGE_EXECUTABLE_RAM_SIZE - Defines the size of the region to which images are allowed to be loaded. Only used if MCUBOOT_UPGRADE_STRATEGY is configured to be RAM_LOAD.

Assemble tool

The assemble.py tool is used to concatenate secure and non-secure binary to a single binary blob. It requires the following definitions:

- SECURE_IMAGE_OFFSET - Defines the offset from the single binary blob base address, where the secure image starts.
- SECURE_IMAGE_MAX_SIZE - Defines the maximum size of the secure image area.
- NON_SECURE_IMAGE_OFFSET - Defines the offset from the single binary blob base address, where the non-secure image starts.

- NON_SECURE_IMAGE_MAX_SIZE - Defines the maximum size of the non-secure image area.

Image tool

The imgtool.py tool is used to handle the tasks related to signing the binary. It requires the following definition:

- S_IMAGE_LOAD_ADDRESS - Defines the address to where the Secure (or combined Secure and Non-secure) image is loaded and is executed from. Only used if MCUBOOT_UPGRADE_STRATEGY is configured to be RAM_LOAD.
- NS_IMAGE_LOAD_ADDRESS - Defines the address to where the Non-secure image is loaded and is executed from. Only used if MCUBOOT_UPGRADE_STRATEGY is configured to be RAM_LOAD and MCUBOOT_IMAGE_NUMBER is greater than 1.

Expose target support for HW components

Services may require HW components to be supported by the target to enable some features (e.g. PS service with rollback protection, etc). The following definitions need to be set in the .cmake file if the target has the following HW components:

- TARGET_NV_COUNTERS_ENABLE - Specifies that the target has non-volatile (NV) counters.

10.1.3 / (TF-M root)

This table describes the structure under the root folder with part of possible folders.

Folder name	Description
bl1	The 1st stage immutable bootloader
bl2	MCUBoot based 2nd stage bootloader
cmake	Cmake files of the build system
config	Configuration system files
docs	The documentation
interface	RoT service API for client calls
lib	The 3rd party libraries
platform	Platform intermedia files
secure_fw	The secure firmware
tools	Tools in scripts for building

10.1.4 platform

The *platform* folder contains SW ports of all *supported platforms* and the files necessary for *adding a new platform*. Please refer to *platform folder document* for more information.

Folder name	Description
include	HAL and platform public headers
ext	Platform ports and related files

platform/ext

This folder can include imported files licensed differently from TF-M's BSD-3 license. More details are in the dedicated document *Details for the platform/ext folder*.

Folder name	Description
accelerator	Supported Crypto HW accelerators
cmsis	A copy of essential CMSIS headers
common	Common HAL implementation
driver	Driver headers for porting
target/<vendor>	Vendor specific folders with ported platforms

Each *vendor* is assigned one folder for usage under *target/*. A *vendor* contributes platform's ports and manages the structure inside it.

10.1.5 secure_fw

This folder contains components needed by secure firmware and the exported interfaces for application, service development and HAL integration.

Folder name	Description
include	Public headers of *secure_fw*
partitions	Default services and SPRTL
spm	PSA FF-M SPM implementation
shared	Sources shared out of SPRTL

The shared sources can be referenced by the building system out of SPRTL. Generally, they are runtime and PSA APIs.

secure_fw/include

This folder holds public headers for external references by clients, services and platforms. Avoid putting private headers, not referenced by other modules in this *include* folder.

secure_fw/partitions

This folder contains default services implemented as partitions and runtime utilities used and provided by TF-M.

Folder name	Description
lib/runtime	The SPRTL sources and intermedia files
<partition_x>	Sources of *partition_x*
<partition_x>/include	RoT Service API headers of *partition_x*
<partition_y>	Sources of *partition_y*
<partition_y>/include	RoT Service API headers of *partition_y*

Here *partition_x* and *partition_y* are examples of RoT services without detailed structure of them.

secure_fw/spm

The SPM is the core component to provide a mechanism for providing secure services complied with PSA FF-M.

Folder name	Description
include	SPM public headers.
core	SPM base functionalities
*ext	Extended SPM functionalities

10.2 SPM Backends

This document briefly introduces the backends of Secure Partition Manager (SPM) in TF-M and how to select one for building.

10.2.1 IPC and SFN

The Firmware Framework M (FF-M)[1],[2] provides two different programming models for Secure Partitions.

- IPC Model

  The Secure Partition processes signals in any order, and can defer responding to a message while continuing to process other signals.

- SFN Model

  The Secure Partition is made up of a collection of callback functions which implement secure services.

Although the programming model is different, they share the same APIs to interact with the SPM. The behaviours of the APIs share the same implementation with slight differences for the two programming models. This is regarded as the `frontend`.

The TF-M runtime implementations behind `frontend` are different. TF-M provides two `backends` correspondingly.

> - IPC backend
>
>   In this backend, the SPM and each Secure Partition have their own execution contexts, which is required to support the IPC model Secure Partitions. This also enables the SPM to provide higher isolation levels. This SPM backend acts like a multiple-process system. It can also adopt SFN model Secure Partitions.
>
> - SFN backend
>
>   The SFN backend provides more efficient executions because it shares a single-thread execution context with all the Secure Partitions. This SPM backend acts like a single library. Therefore, it can only adopt SFN model Secure Partitions. And it does not support higher isolation levels. On the other hand, it consumes less memory compared to the IPC backend.

[1] FF-M v1.0 Specification

[2] FF-M v1.1 Extension

The following table summaries the relationships between SPM backends, Secure Partition models and isolation levels.

SPM backend	Supported Partition model	Supported Isolation Level
SFN	SFN Partition	1
IPC	IPC and SFN Partition	1, 2 and 3

10.2.2 Implementation Recommendations

If an implementation doesn't contain any IPC model Secure Partition and only requires isolation level 1, then it is recommended to select the SFN backend to optimize memory consumption and execution performance.

If an implementation contains any IPC model Secure Partition or requires isolation level 2 or 3, then the IPC backend is required.

10.2.3 TF-M Configuration Switches

In the TF-M build system, the `CONFIG_TFM_SPM_BACKEND` configuration is used to select the backend of SPM. The valid values are `SFN` and `IPC`.

```
-DCONFIG_TFM_SPM_BACKEND=SFN
```

If `CONFIG_TFM_SPM_BACKEND` is not set, then `IPC` is the default value.

10.2.4 References

10.3 Non-secure Client Extension Integration Guide

This document introduces TF-M Non-secure Client Extension (NSCE) and how to integrate it with Non-secure Processing Environment (NSPE) RTOS.

10.3.1 What is NSCE for

Besides the secure services provided via PSA APIs, there are two interactions between TF-M and NSPE RTOS.

- Non-secure context management in TF-M

  When a NS task calls a secure service, a context is maintained in TF-M. If TF-M supports multiple secure service calls, the context needs to be loaded and saved with the corresponding NS task when the RTOS kernel (the kernel) does scheduling.

- Non-secure client ID (NSID) management

  As per PSA Firmware Framework specification, NSID is required for a secure service call from the NSPE. The NSID can be managed by either SPM (the same NSID must be used for all connections) or the NSPE RTOS. For the latter case, the NSPE RTOS must provide the NSID for each connection.

NSCE is implemented to support the interactions above.

- Provide the interface to NSPE RTOS for managing the context in TF-M.
- Provide a mechanism to NSPE RTOS for providing the NSID for each connection.

10.3.2 NSCE component diagram

Figure 2:: Non-secure Client Extension Component Diagram

This diagram shows the components of NSCE.

- NSCE interface: The NSCE API interface to NSPE RTOS.
- NS Client Context Manager: It is the internal component of SPM that is used to manage the NS client context. It is not accessible to NSPE.

10.3.3 Group-based context management

NSCE introduces a group-based context management for NS clients. The purpose is to support diverse use scenarios in the IoT world.

For rich-resource devices, TF-M could assign separate contexts for each connection. For resource-constrained devices, TF-M may support very limited context slots. Multiple NS connections could share the same context provided they have mutual exclusion access to the secure services.

To support this flexibility, a group ID (gid) and a thread ID (tid) are specified for the connection of the NS client. NSCE allocates only one context for a given group. Threads in the same group share the context.

The gid and tid are specified by NSPE RTOS via NSCE interface.

10.3.4 NSCE interface

NSCE defines a set of APIs as the interface for NSPE RTOS to manage the context in TF-M. NSID is provided as an input parameter of *tfm_nsce_load_ctx()*.

NSCE APIs are typically called by the kernel and must be called from the non-secure handler mode.

```
uint32_t tfm_nsce_init(uint32_t ctx_requested)
```

This function should be called before any other NSCE APIs below to do non-secure context initialization in NSCE for the calling NS entity. *ctx_requested* is the number of the contexts requested by the NS entity. The number of assigned context will be returned. It may be equal to or smaller than the requested context number based on the current available resource in TF-M. If *ctx_requested* is *0*, then the maximum available context number will be assigned and returned. If the initialization is failed, *0* should be returned.

Note: As TF-M only supports one context for now, so the return value is always *1* if no error. Currently, it is safe to skip calling *tfm_nsce_init()*. But, for future compatibility, it is recommended to do so.

```
uint32_t tfm_nsce_acquire_ctx(uint8_t group_id, uint8_t thread_id)
```

This function allocates a context for the NS client connection. The *gid* and *tid* are the input paramemters. A token will be returned to the NSPE if TF-M has an available context slot. Otherwise, *TFM_NS_CLIENT_INVALID_TOKEN* is returned. It is the responsibility of NSPE RTOS to assign gid and tid for each NS client.

```
uint32_t tfm_nsce_release_ctx(uint32_t token)
```

This function tries to release the context which was retrieved by *tfm_nsce_acquire_ctx()*. As the context may be shared with other threads within the same group, the context is only freed and back to the available context pool after all threads in the same group release the context.

```
uint32_t tfm_nsce_load_ctx(uint32_t token, int32_t nsid)
```

This function should be called when NSPE RTOS schedules in a NS client. *token* is returned by *tfm_nsce_acquire_ctx()*. *nsid* is the non-secure client ID used for the following PSA service calls.

The assignment of NSID is managed by the NSPE RTOS. It is not required to use the same NSID for a NS client when calling this function each time. This allows the NS client changing its NSID in the lifecycle. For example, the provisioning task may need to switch NSID to provision the keys for different NS clients created afterwards.

```
uint32_t tfm_nsce_save_ctx(uint32_t token)
```

This function should be called when NSPE RTOS schedules out a NS client. The input parameter *token* is returned by *tfm_nsce_acquire_ctx()*. After the context is saved, no secure service call can be made from NSPE until a context is loaded via *tfm_nsce_load_ctx()*.

10.3.5 NSCE integration guide

Enable NSCE in TF-M

To enable NSCE in TF-M, set the build flag *TFM_NS_MANAGE_NSID* to *ON* (default *OFF*).

Support NSCE in an RTOS

For supporting NSCE in an RTOS, the integrator needs to do the following major work:

- Integrate the NSCE API calls into the NS task lifecycle. For example, creating/scheduling/destroying a NS task.
- Manage the assignment for *gid*, *tid* and NSID.

The typical steps are listed below:

- Before programming, the integrator should plan the group assignment for the NS tasks that need to call secure services. If the number of tasks is less than or equal to TF-M non-secure context slots, then different gid can be assigned to each task for taking dedicated context in TF-M. Otherwise, the integrator need to think about grouping the tasks to share the context.
- In the kernel initialization stage, it calls *tfm_nsce_init()* with requested context number to initialize the non-secure context in TF-M. The actual allocated context number will be returned. *0* means initialization failed. The kernel could use different group assignment sets according to the context number allocated to it. TF-M only supports one context for now.
- The kernel calls *tfm_nsce_acquire_ctx()* when creating a new task. This should be done before the new task calls any secure service. A valid token returned should be saved as part of the task context in NSPE RTOS.
- When the kernel schedules in a task with a valid *token* associated, *tfm_nsce_load_ctx()* should be called before resuming the execution of that task. The NSID is specified by the kernel through the *nsid* parameter. The mapping between NSID and task is managed by the kernel. *tfm_nsce_load_ctx()* can be called multiple times without calling *tfm_nsce_save_ctx()* for switching the NSID for the same task (same tid and gid).

- *tfm_nsce_save_ctx()* should be called if the current task has a valid *token* before being switched to another task. Calling *tfm_nsce_load_ctx()* for another task before saving the current context may result in NS context lost in TF-M for the running task.
- When the task is terminated, destroyed or crashed, the kernel should call *tfm_nsce_release_ctx* to make sure the associated resource is back to the pool in TF-M.

Integration example

Figure 3:: RTOS/NSCE integration example

This is the software module diagram of a typical RTOS/NSCE integration example.

- Built-in Secure Context Manager: An RTOS may have an existing built-in secure context manager with a group of secure context management APIs defined. Let's take RTX which uses Armv8-M TrustZone APIs as the example.

Note: RTOS may define the NS task context in the secure side as "secure context". It is the same thing as the "non-secure context" (context for a secure service call from NS client) from TF-M's point of view.

- Shim Layer/Secure Context Manager: If the RTOS has a "Built-in Secure Context Manager", then a shim layer should be provided to translate the built-in API calls into the NSCE API calls. A reference shim layer for RTX TrustZone APIs is here.

  If the RTOS has no existing one, then a "Secure Context Manager" should be implemented based on the NSCE APIs. You can refer to the shim layer example above for the implementation. The timing of calling NSCE APIs in the kernel is introduced in the *Support NSCE in an RTOS* section above.
- NSID Manager: If NSPE RTOS manages the NSID, then this module is used by the secure context manager or shim layer to manage the NSID assignment for the NS tasks. The assignment is implementation defined. A task name based NSID manager is provided as a reference.

Integration notes

- *gid*: It is a *uint8_t* value (valid range is 0 - 255). So, maximum 256 groups (NSCE context slots) are supported by the NSCE interface. TF-M only supports single context for now. So, it is recommended to use single group ID at this stage.
- *tid*: It is a *uint8_t* value (valid range is 0 - 255). Thread ID is used to identify a NS client within a given group. *tid* has no special meaning for TF-M. So, usually the kernel only needs to ensure a NS task has a unique *tid* within a group.
- *gid* and *tid* management: It is the responsibility of NSPE RTOS to manage the assignment of *gid* and *tid*. Based on the explaination above, the *gid* could be assigned as a constant value. And, the *tid* can be increased globally when calling *tfm_nsce_acquire_ctx()* for a new task. Just notice *tid* may overflow.
- NSID management: It is the responsibility of NSPE RTOS to manage the assignment of the NSID for each task. It is highly recommended that a NS client uses the same NSID following a reboot or update. The binding of a NS client to a specific NSID will ensure the correct access to the assets. For example, the data saved in the protected storage.
- Integrate with the existing secure context management APIs of NSPE RTOS: The NSCE APIs are designed to be compatible with most known existing secure context management APIs. A shim layer is needed to translate the API calls. See the integration example above.

- *tfm_nsce_acquire_ctx()* must be called before calling *tfm_nsce_load_ctx()*, *tfm_nsce_save_ctx()* or *tfm_nsce_release_ctx()*.
- *tfm_nsce_release_ctx()* can be called without calling *tfm_nsce_save_ctx()* ahead.

10.4 Generic OS migration from Armv7-M to Armv8-M architecture

The purpose of this document is to list a set of requirements needed for migrating a generic OS kernel running on Armv7-M to the Armv8-M architecture.

10.4.1 List of requirements

- If the same OS codebase is used for both Secure and Non Secure builds, it is suggested to put specific code targeted to the Non Secure build under a compile time switch, e.g. `#if (DOMAIN_NS == 1U)`. The OS build system in this case needs to be amended accordingly to support this new switch.
- If the OS implements stack limit checking, the `PSPLIM` register needs to be initialized and properly handled during thread context switch operations.
- If the OS manipulates directly the Link Register, the default Link Register value used in Handler mode transitions needs to be differentiated between Secure and Non Secure builds, i.e. `0xFD` and `0xBC`, respectively.
- If the OS manages the non-secure client identification, please check the *Non-secure Client Extension Integration Guide*.

10.5 Floating-Point Support

TF-M adds several configuration flags to control Floating point (FP)[1] support in TF-M Secure Processing Environment (SPE) and Non Secure Processing Environment (NSPE).

- Support FP in SPE or NSPE.
- Support FP Application Binary Interface (ABI)[2] types: software, hardware. SPE and NSPE shall use the same FP ABI type.
- Support lazy stacking enable/disable in SPE only, NSPE is not allowed to enable/disable this feature.
- Support GNU Arm Embedded Toolchain[3]. `GNU Arm Embedded Toolchain 10.3- 2021.10` and later version shall be used to mitigate VLLDM instruction security vulnerability[4].
- Support both IPC[5] and SFN[11] models in TF-M.
- Support Armv8-M mainline.

[1] High-Performance Hardware Support for Floating-Point Operations
[2] Float Point ABI
[3] GNU Arm Embedded Toolchain
[4] VLLDM instruction Security Vulnerability
[5] Arm® Platform Security Architecture Firmware Framework 1.0
[11] FF-M v1.1 Extension

- Support isolation level 1,2,3.
- Support Arm Compiler for Embedded[10]. `Arm Compiler for Embedded 6.17` and later version shall be used to mitigate VLLDM instruction security vulnerability[Page 199, 4].
- Does not support use FPU in First-Level Interrupt Handling (FLIH)[6] at current stage.

Please refer to Arm AN521 or AN552 platform as a reference implementation when you enable FP support on your platforms.

Note: Alternatively, if you intend to use FP in your own NSPE application but the TF-M SPE services that you enable do not require FP, you can set the CMake configuration `CONFIG_TFM_ENABLE_CP10CP11` to `ON` and **ignore** any configurations described below.

Note: FPU test issue has not been fixed yet on Musca-S1[7]. When running FPU tests on Musca-S1, secure thread fails to trigger secure interrupt. FPU test is disabled by default on Musca-S1 until the issue is fixed.

Note: `GNU Arm Embedded Toolchain 10.3-2021.10` may have issue that reports `'-mcpu=cortex-m55' conflicts with '-march=armv8.1-m.main'` warning[8]. This issue has been fixed in the later version.

10.5.1 FP ABI type for SPE and NSPE

FP design in Armv8.0-M[9] architecture requires consistent FP ABI types between SPE and NSPE. Furthermore, both sides shall set up CPACR individually when FPU is used. Otherwise, No Coprocessor (NOCP) usage fault will be asserted during FP context switch between security states.

Secure and non-secure libraries are compiled with `COMPILER_CP_FLAG` and linked with `LINKER_CP_OPTION` for different FP ABI types. All those libraries shall be built with `COMPLIER_CP_FLAG`.

If FP ABI types mismatch error is generated during build, pleae check whether the library is compiled with `COMPILER_CP_FLAG`. Example:

```
target_compile_options(lib
    PRIVATE
        ${COMPILER_CP_FLAG}
)
```

[10] Arm Compiler for Embedded
[6] *Secure Interrupt Integration Guide*
[7] Musca-S1 Test Chip Board
[8] GCC Issue on '-mcpu=cortex-m55' conflicts with '-march=armv8.1-m.main' Warning
[9] Armv8-M Architecture Reference Manual

10.5.2 CMake configurations for FP support

The following CMake configurations configure `COMPILER_CP_FLAG` in TF-M SPE.

- `CONFIG_TFM_ENABLE_FP` is used to enable/disable FPU usage.

CONFIG_TFM_ENABLE_FP	FP support
off (default)	FP disabled
on	FP enabled

Note: `CONFIG_TFM_FLOAT_ABI` depends on `CONFIG_TFM_ENABLE_FP`. If `CONFIG_TFM_ENABLE_FP is set ``CONFIG_TFM_FLOAT_ABI` is automatically set to `hard`.

Note: If you build TF-M SPE with `CONFIG_TFM_ENABLE_FP=on` and provide your own NSPE application, your own NSPE **must** take care of enabling floating point coprocessors CP10 and CP11 on the NS side to avoid aforementioned NOCP usage fault.

- `CONFIG_TFM_LAZY_STACKING` is used to enable/disable lazy stacking feature. This feature is only valid for FP hardware ABI type. NSPE is not allowed to enable/disable this feature. Let SPE decide the secure/non-secure shared setting of lazy stacking to avoid the possible side-path brought by flexibility.

CONFIG_TFM_LAZY_STACKING	Description
0FF	Disable lazy stacking
ON (default)	Enable lazy stacking

- `CONFIG_TFM_FP_ARCH` specifies which FP architecture is available on the target, valid for FP hardware ABI type.

  FP architecture is processor dependent. For GNUARM compiler, example value are: auto, fpv5-d16, fpv5-sp-d16, etc. For armclang, example value are: none, softvfp, fpv5-d16, fpv5-sp-d16, etc.

  This parameter shall be specified by platform in preload.cmake. Please check compiler reference manual and processor hardware manual for more details to set correct FPU configuration for platform.

- `CONFIG_TFM_FP_ARCH_ASM` specifies the target FPU architecture name shared by Arm Compiler armasm and armlink. It is only used in the `--fpu=` argument by Arm Compiler and shall be aligned with `CONFIG_TFM_FP_ARCH`.

  FP architecture is processor dependent. For armasm and armlink, example value are: SoftVFP, FPv5_D16, etc.

  This parameter shall be specified by platform in preload.cmake. Please check compiler reference manual and processor hardware manual for more details to set correct FPU configuration for platform.

Reference

10.6 Secure Interrupt Integration Guide

10.6.1 Introduction

This document describes how to enable an interrupt in TF-M. The target audiences are mainly platform integrators and Secure Partition developers.

This document assumes that you have read the PSA Firmware Framework (FF-M) v1.0[1] and the FF-M v1.1 extensions[2] thus have knowlege on the terminologies such as Secure Partitions and manifests.

10.6.2 Interrupt Handling Model

TF-M supports the two interrupt handling models defined by FF-M:

- First-Level Interrupt Handling (FLIH)

  In this model, the interrupt handling is carried out immediately when the interrupt exception happens.

  The interrupt handling can optionally set an interrupt signal for the Secure Partition Thread to have further data processing.

- Second-Level Interrupt Handling (SLIH)

  In this model, the interrupt handling is deferred after the interrupt exception. The handling occurs in the Secure Partition Thread thus is subject to scheduling.

The FLIH supports handling an interrupt in a bounded time, but very limited APIs are allowed in the FLIH handling because the handling occurs in an special exception context.

The SLIH is deferred and subject to scheduling but all Secure Partition APIs are allowed as the SLIH handling is in the Secure Partition Thread.

Both the FLIH and the SLIH can be used by Secure Partitions which conform to Firmware Framework v1.1.

While the SLIH is the only supported model for Secure Partitions which conform to Firmware Framework v1.0.

Please refer to chapter 6.2 of FF-M v1.1[Page 202, 1] for more details on the interrupt handling models.

10.6.3 Enabling an Interrupt

To enable an interrupt, you need to do the following:

- Binding the interrupt to a Secure Partition.
- Granting the Secure Partition access permissions to the device of the interrupt.
- Initializing the interrupt.
- Integrating the interrupt handling function

[1] FF-M v1.0 Specification

[2] FF-M v1.1 Extension

TF-M has two Test Partitions as good examples for both FLIH[3] and SLIH[4]. See also *Enabling the Interrupt Tests* on how to integrate them to platforms.

Binding an Interrupt to a Secure Partition

To bind an interrupt to a Secure Partition, you need to add an item to the `irqs` attribute of the Secure Partition manifest. `irqs` is a list of Interrupt Request (IRQ) assigned to the Secure Partition.

Secure Partitions are not allowed to share IRQs with other Secure Partitions.

Different Firmware Framework versions have different definitions of manifest.

FF-M v1.0

Here is an example manifest of Secure Partitions conform to Firmware Framework version 1.0:

```
{
  "irqs": [
    {
      "source": 5,
      "signal": "DUAL_TIMER_SIGNAL"
    },
    {
      "source": "TIMER_1_SOURCE",
      "signal": "TIMER_1_SIGNAL"
    }
  ]
}
```

- source

  Required, Unique.

  The `source` is a string that identifies the interrupt source. It can be a valid exception number or a symbolic name defined in platform codes.

- signal

  Required, Unique.

  The `signal` attribute is a symbolic name used by TF-M to identify which interrupt is asserted. It is also used by the Secure Partition to receive the interrupt signal by calling `psa_wait` for interrupt handling.

  It is defined in the Secure Partition header file <`psa_manifest/manifestfilename.h`> generated by TF-M:

  ```
  #define signal VALUE
  ```

The interrupt handling model is SLIH by default as it is the only supported one for FF-M v1.0.

[3] https://git.trustedfirmware.org/TF-M/tf-m-tests.git/tree/tests_reg/test/secure_fw/suites/spm/irq/service/tfm_flih_test_service

[4] https://git.trustedfirmware.org/TF-M/tf-m-tests.git/tree/tests_reg/test/secure_fw/suites/spm/irq/service/tfm_slih_test_service

FF-M v1.1

Here is an example manifest of Secure Partitions conform to Firmware Framework version 1.1:

```
{
  "irqs": [
    {
      "source"  : "TIMER_1_SOURCE",
      "name"    : "TIMER_1",
      "handling": "FLIH"
    },
    {
      "source"  : 5,
      "name"    : "DUAL_TIMER",
      "handling": "SLIH"
    }
  ]
}
```

- source

  The source is the same as the one in Firmware Framework Version 1.0.

- name

  Required, Unique.

  The name is used to construct the following two elements:

  - the interrupt signal symbol: {{name}}_SIGNAL, the equivalent of signal in FF-M v1.0
  - the FLIH Function for handling FLIH IRQs provided by Secure Partition:

    psa_flih_result_t {{name}}_flih(void);

    It is also declared in <psa_manifest/manifestfilename.h>.

- handling

  Required.

  The handling attribute specifies the interrupt handling model and must have one of the following values:

  - FLIH - First-Level Interrupt Handling
  - SLIH - Second-Level Interrupt Handling

Granting Permissions to Devices for Secure Partitions

A secure partition shall be granted two parts of permissions to access a device. One is the Memory Maped I/O (MMIO) region of the device. The other is the driver codes to access the device.

The MMIO Regions

You need to declare the MMIO region in the `mmio_regions` attributes in the Secure Partition manifest, to enable the Secure Partition to access it.

An MMIO region can be described as either `numbered_region` or `named_region`. A numbered region consists of a `base` address and a `size`. A named region consists of a string `name` to describe the region.

Here is an example of named region:

```
{
  "mmio_regions": [
    {
      "name": "TFM_PERIPHERAL_TIMER0",
      "permission": "READ-WRITE"
    }
  ]
}
```

- name

  Required.

  The `name` attribute is a symbolic name defined by platforms. It is a pointer to structure instance that usually includes the base address and size of the region and some other platform specific attributes that are required to set up permissions to the MMIO region.

  The structure is defined by platforms and the name must be `struct platform_data_t`.

- permission

  Required.

  The `permission` attribute must have one of the following values:

  - `READ-ONLY`
  - `READ-WRITE`

The Device Drivers

To give permissions of devices drivers to Secure Partitions, it is recommended to put the driver codes to the Partition's CMake library:

```
target_sources(some_partition_lib
    PRIVATE
        some_driver_code.c
)
```

Initializing the Interrupts

Platforms must define an interrupt initialization function for each Secure interrupt.

The prototype of the function is:

```
enum tfm_hal_status_t {source_symbol}_init(void *p_pt,
                                           const struct irq_load_info_t *p_ildi)
```

The `{source_symbol}` is:

- `irq_{source}`, if the `source` attribute of the IRQ in Partition manifest is a number
- Lowercase of `source` attribute, if `source` is a symbolic name

For example if the manifest declares `"source": 5`, then the function name is `irq_5_init`. If the manifest declares `"source" : "TIMER_1_IRQ"` then the function name is `timer_1_irq_init`.

The function will be called by the Framework automatically during initialization. The function can be put in any source file that belongs to SPM, for example a `tfm_interrupts.c` added to the `tfm_spm` CMake target.

The initialization of an interrupt must include:

- setting the priority
- ensuring that the interrupt targets the Secure State.
- saving the interrupt information

Setting Priority

The priority of external interrupts must be in the following range: `(0, N / 2)`, where `N` is the number of configurable priorities. Smaller values have higher priorities.

For example if the number of configurable priority of your interrupt controller is 16, you must use the priorities in range `(0, 8)` only, boundaries excluded.

Note that these are not the values set into the interrupt controllers. Different platforms may have different values for those priorities. But if you use the `NVIC_SetPriority` function provided by CMSIS to set priorities, you can pass the values directly.

Platforms have the flexibilities on the assignment of priorities.

Targeting Interrupts to Secure

In single core systems, platform integrators must ensure that the Secure interrupts target to Secure State by setting the Interrupt Controller.

In multi-core systems, this might be optional.

Saving the Interrupt Information

The initialization function is called during Partition loading with the following information:

- p_pt - pointer to Partition runtime struct of the owner Partition
- p_ildi - pointer to irq_load_info_t struct of the interrupt

Platforms must save the information for the future use. See *Integrating the Interrupt Handling Function* for the usage.

The easiest way is to save them in global variables for each interrupt. TF-M provides a struct for saving the information:

```
struct irq_t {
    void                         *p_pt;
    const struct irq_load_info_t *p_ildi;
};
```

Integrating the Interrupt Handling Function

TF-M provides an interrupt handling entry for Secure interrupts:

```
void spm_handle_interrupt(void *p_pt, const struct irq_load_info_t *p_ildi)
```

The p_pt and p_ildi are the information passed to interrupt initialization functions and saved by platforms.

Platforms should call this entry function in the interrupt handlers held in Vector Table with the information saved by the interrupt initialization functions. If the information is saved as global variables, then the interrupt handlers can be put in the same source file that contains the initialization functions.

Here is an example:

```
void TFM_TIMER0_IRQ_Handler(void) /* The handler in Vector Table */
{
    spm_handle_interrupt(p_timer0_pt, p_tfm_timer0_irq_ldinf);
}
```

10.6.4 Enabling the Interrupt Tests

TF-M provides test suites for FLIH and SLIH interrupts respectively. They are disabled by default.

Note: FLIH interrupt test and SLIH interrupt test share the same timer TFM_TIMER0_IRQ thus cannot be enabled at the same time.

To enable the tests, please follow steps in the previous sections. In addition, you need to implement the following APIs of timer control:

- void tfm_plat_test_secure_timer_start(void)
- void tfm_plat_test_secure_timer_clear_intr(void)
- void tfm_plat_test_secure_timer_stop(void)

You shall also select the following flags in platform specific config.cmake to indicate that FLIH and SLIH interrupt tests are supported respectively.

- PLATFORM_FLIH_IRQ_TEST_SUPPORT: platform implements support of FLIH interrupt tests

- PLATFORM_SLIH_IRQ_TEST_SUPPORT: platform implements support of SLIH interrupt tests

The following configurations control SLIH and FLIH interrupt tests:

- TEST_NS_FLIH_IRQ
- TEST_NS_SLIH_IRQ

They can be enabled via build command line or via TEST_NS.

10.6.5 Migrating to Firmware Framework v1.1

Please refer to Migrating Secure Partitions to version 1.1 of FF-M v1.1[1] .

10.6.6 References

10.7 Platform Provisioning

TF-M stores any data that should be provisioned at the factory in OTP memory. The default is that this OTP memory is actually implemented using on-chip flash, the same that is used to implement the ITS service.

If the lifecycle state is in the TFM_SLC_ASSEMBLY_AND_TEST[1] state (which is the default for non-provisioned boards), then TF-M will attempt to provision the: - HUK instead of booting. It will read the data from the assembly_and_test_prov_data struct, and will then provision it to OTP. The lifecycle state will then transition to TFM_SLC_PSA_ROT_PROVISIONING[1].

If the lifecycle state is in the TFM_SLC_PSA_ROT_PROVISIONING[1] state, then TF-M will attempt to provision the:

- IAK
- boot seed
- implementation id
- certification reference
- bl2 ROTPKs (of which there are up to 4)
- entropy seed

Once all of these have been loaded from the psa_rot_prov_data struct and provisioned to OTP, the LCS will transition to TFM_SLC_SECURED[1]. Note that this provisioning step will be run immediately after the TFM_SLC_ASSEMBLY_AND_TEST[1] provisioning stage if the lifecycle transition completed successfully.

[1] For the definitions of these lifecycle states, please refer to the Platform Security Model https://developer.arm.com/documentation/den0128/0100/

10.7.1 Provisioning development hardware

If TFM_DUMMY_PROVISIONING is enabled in the cmake config (as it is by default), a set of dummy keys / data will be provisioned. The dummy IAK matches the IAK tested by the TF-M tests, and the dummy bl2 ROTPKs match the dummy bl2 keys used by default. TFM_DUMMY_PROVISIONING _MUST_ not be used in production hardware, as the keys are insecure.

10.7.2 Provisioning production hardware

For provisioning of real hardware, firstly TFM_DUMMY_PROVISIONING must be disabled. Then it is required to inject the keys into RAM so they populate the assembly_and_test_prov_data and psa_rot_prov_data structs, at the beginning of the TF-M boot. These structs each require a magic value to be set to be accepted by the provisioning code, which is detailed in platform/ext/common/provisioning.c. Two suggestions for how to do this are:

- Attach a debugger, and inject the values into RAM.
- Flash an image that contains the required data. Care must be taken with this approach that the keys are not left in RAM after provisioning, so a different image (without provisioning data embedded) must be flashed afterwards, without erasing the OTP flash area.

Provisioning on CryptoCell-312 enabled platforms

On boards that have a CC312 accelerator, and that have the default flash-backed OTP disabled by setting PLATFORM_DEFAULT_OTP=OFF in cmake, the CC312 OTP will be used as a backing for the OTP HAL.

Due to the CC312 requiring a power-cycle to transition LCS, you will be prompted to manually power-cycle the board between provisioning stages.

Boards with real OTP memory cannot be reprovisioned - care should be taken that the data being provisioned is the desired data.

Platform-specific OTP backing

If a platform has a medium that is suitable for storing data with OTP semantics (Where a bit cannot transition from a 1 to a 0), such as physical OTP memory, then it can provide a backing for the OTP HAL by implementing the methods described in tfm_plat_otp.h.

10.8 Adding a new platform

10.8.1 Porting TF-M to a New Hardware

The purpose of this document is to provide a guide on how to integrate TF-M with another hardware platform. This document will give general guidance on how to port a platform on the TF-M build system and which interfaces must exist on the platform for TF-M (S and NS) to run on this new platform.

Prerequisites

Building environnement

Make sure you have a working build environnement and that you can build TF-M on AN521 following the *Build instructions*.

Toolchains and software requirements

Please follow the Getting started guide.

CMSIS Drivers

The TF-M stack requires at least two CMSIS HAL implementations:

- USART
- FLASH

Porting flow

In a nutshell, this should be a 6 iterative steps process:

1. Adding all the mandatory files and expected objects/functions declarations
2. Booting and configuring the core(s)
   - startup(s) code and SystemInit
3. Adding the USART drivers
   - CMSIS HAL
4. Adding the FLASH drivers
   - CMSIS HAL
5. Enabling/Configuring/Disabling features including templated features
   - E.G. NV Counters, attestation, crypto keys. . . .
6. Adding the optional platform SVC handling

   Some platforms may have their own SVC requests in addition to the TF-M built-in ones.
7. Running the regression tests
   - See *Running TF-M on Arm platforms* as an example

File architecture

The platform selection when building TF-M is set via the CMake variable TFM_PLATFORM. This variable holds part of the path to the platform. When using `-DTFM_PLATFORM=arm/mps2/an521` or `-DTFM_PLATFORM=an521` TF-M build system will look for the platform in <TF-M ROOT>/platform/ext/target/arm/mps2/an521. Therefore all hardware dependent code for your platform should go to <TF-M ROOT>/platform/ext/target/.

platform/ext/target

This folder contains a first level of board vendor (such as ARM, STM, NXP, Cypress ....), each folder will usually contain a second level for each board. This second level is not mandatory.

> platform/ext/target/<vendor>/[<board name>/]

From now on this will be referred to as the `platform folder`.

platform/ext/common

This folder contains files and folder commons to the platforms, such as the shims to the CMSIS drivers. It also contains the scatter files that can be used for the bl2, tfm_s, tfm_ns partitions.

This folder also contains another folder named template. The latter contains example implementations that are used for platforms by default, but which can be altered or replaced if other functionality is required.

name	description
PLATFORM_DEFAULT_ATTEST_HAL	Use the default implementation of the attestation HAL (default True)
PLATFORM_DEFAULT_NV_COUNTERS	Use the default implementation of the counters in NV (default True)
PLATFORM_DEFAULT_CRYPTO_KEYS	Use the default implementation of crypto keys (default True)
PLATFORM_DEFAULT_ROTPK	Use the default implementation of the RoT Public Key (default True)
PLATFORM_DEFAULT_IAK	Use the default implementation of the initial attestation key (default True)
PLATFORM_DEFAULT_UART_STDOUT	Use the default implementation of the uart for stdout output (default True)
PLATFORM_DEFAULT_NV_SEED	Use the default implementation of the NV seed in the RNG (default True)
PLATFORM_DEFAULT_OTP	Use the default implementation of the OTP (default True)

Platform Folder

Description

Depending on the level of integration you want with TF-M some files or information will be mandatory for the build system to build working firmware.

Please note that platform folder provides source for building both *SPE* and *NSPE* parts. The SPE builds directly from the source tree while files necessary for NSPE platform support are installed to `<Artifact folder>` for building TF-M application as decribed in the *Build instructions*.

Questions to be answered:

- Will the platform use MCUboot as the second stage bootloader?

  BL2/MCUboot provides a secure bootloader that enables simple software upgrades.

  This optional second stage bootloader is set-up via the bl2 target in the CMakelists.txt file (see below).

- Will the platform support the Non-Secure world application?

  A platform can be designed to only support the secure world, in which case we would refer to it as a secure enclave. TF-M build system allows the developer to strip all Non-Secure world related code out of the final image. Most platforms, and especially the ones intended to be generic or to have a Non-Secure application will require Non-Secure world support. In that case a platform shall instruct build system on the file set for exporting to Non-Secure world.

- How does the non-secure world communicate with the secure world?

  TF-M supports running the non-secure world on the same CPU as the secure world, communicating via TrustZone or running the non-secure world on a separate CPU, communicating via a mailbox. The platform is responsible for configuring toolchains with correct CPU and architecture related features for secure and non-secure worlds.

  The architecture for secure world is configured in the cpuarch.cmake file (see below).

- How does the FLASH need to be split between worlds?

  The flash split is very dependent on the support of BL2 and NS world. When porting a new platform, one shall arrange enough flash size for each of them.

  If supporting upgrades (via MCUboot), additional flash area will be required to store the updates before upgrading the whole system.

- How does the RAM need to be split between worlds?

  The RAM split is very dependent on the support of the NS world.

  If you’re not porting the platform for a specific project but are enabling the Non-Secure world, you should ensure that you leave enough RAM available for it to run.

Note: TF-M S world size in RAM and Flash varies greatly with different build options.

TF-M project provides metrics of the S world size for existing platforms, which may help to get a rough guide to the sizes needed.

Files

CMakeLists.txt :

(MANDATORY)

This is the entry point for the build system to build your platform on the secure side and also export files to build Non-Secure side.

it must:

- Add a folder to the target platform_region_defs. [*PLATFORM_REGION_DEFS*]

  This folder will contain two files *flash_layout.h* and *region_defs.h*

- Add scatter files to the bl2 and tfm_s targets. [*SCATTER*]

Please note that TF-M provides a common scatter file for the bl2, tfm_s and tfm_ns targets, which can be used in most cases.

- Add startup files to the bl2 and tfm_s targets. [*STARTUP*]
- Add required sources and includes for the bl2 and tfm_s targets [*SOURCES_INCLUDES*]
- Install all files required for building the platform in the Non-secure application [*INSTALL*]

The installation section expands the common installation script with the platform specific files. The following predefined variables are availble to address the respective subfolders of the target `<Artifact folder>`.

name	description
INSTALL_INTERFACE_INC_DIR	interface/include - interface header files
INSTALL_INTERFACE_SRC_DIR	interface/src - interface source files
INSTALL_INTERFACE_LIB_DIR	interface/lib - interface libraries
INSTALL_IMAGE_SIGNING_DIR	image_signing tools and files
INSTALL_CMAKE_DIR	CMake modules for Non-secure app build
INSTALL_PLATFORM_NS_DIR	NS platform source files

config.cmake:

(MANDATORY)

This file is used to setup default build configurations for TF-M and platform configurations which have fixed values depending on hardware and software supportness. These configurations should be set as normal CMake variables while others are cache variables.

The platform configurations in the below table are required.

name	description
CONFIG_TFM_USE_TRUSTZONE	Use TrustZone to transition between NSPE and SPE on the same CPU
TFM_MULTI_CORE_TOPOLOGY	NSPE runs on a separate CPU to SPE

The platform configurations in the below table control optional features which rely on platform specific implementation. These features are disabled by default. Platforms shall implement corresponding functionalities and explicitly set the configuration to enable the feature.

name	description
PLATFORM_HAS_ISOLATION_L3_SUPPORT	Whether the platform has isolation level 3 support
PLATFORM_HAS_FIRMWARE_UPDATE_SUPPORT	Wheter the platform has firmware update support
PSA_API_TEST_TARGET	The target platform name of PSA API test
PLATFORM_SVC_HANDLERS	Whether the platform has specific SVC handling

For build configurations, please refer to `config_base.cmake`.

[*config_cmake*]

cpuarch.cmake:

(MANDATORY)

This file contains hardware information such as the main processor and architecture of the SPE CPU. On single-core platforms, it should be installed to <`Artifact folder`> for NSPE build. On multi-core platforms, two cpuarch.cmake files should be added.

- a SPE specific `cpuarch.cmake` used in SPE build
- an NSPE one which should be installed to <`Artifact folder`> with filename `cpuarch.cmake` for NSPE build. See *ns/cpuarch_ns.cmake*.

name	description
TFM_SYSTEM_PROCESSOR	The SPE Processor the platform is using
TFM_SYSTEM_ARCHITECTURE	The architecture of the processor
CONFIG_TFM_FP_ARCH	The Float Point architecture flag for toolchain
CONFIG_TFM_FP_ARCH_ASM	The Float Point architecture flag for assembly code

tests/tfm_tests_config.cmake:

(OPTIONAL)

This file contains platform-specific config options for TF-M regression tests. The `tests` folder should installed to <Artifact folder>/platform for NSPE build. Here are some examples.

<table>
<tr><th>name</th><th>description</th></tr>
<tr><td colspan="2">PLATFORM_SLIH_IRQ_TEST_SUPPORT | Whether the platform has SLIH test support</td></tr>
<tr><td colspan="2">PLATFORM_FLIH_IRQ_TEST_SUPPORT | Whether the platform has FLIH test support</td></tr>
</table>

tests/psa_arch_tests_config.cmake:

(OPTIONAL)

This file contains platform-specific config options for PSA API tests. Here are some examples.

name	description
PSA_API_TEST_TARGET	The target platform name of PSA API test

startup files:

(MANDATORY)

These files (one for BL2, one for S, one for NS) are the expected startup files. The reset handler should call SystemInit and then should end up calling __START which should be defined as _start if not defined elsewhere.

flash_layout.h:

(MANDATORY)

This file can be anywhere in the platform folder, usually in a sub folder named `partition`. TF-M doesn't provide a template for this file, common practice is to copy it from another platform (e.g. arm/mps2/an521) and update the following entries.

Note: all size are in bytes

name	description	Requisiteness
FLASH_S_PARTITION_SIZE	Size of the Secure partition in flash	Yes
FLASH_NS_PARTITION_SIZE	Size of the Non-Secure partition in flash	if tfm_ns is built
FLASH_AREA_IMAGE_SECTOR_SIZE	Size of the flash sector	if bl2 is built
FLASH_TOTAL_SIZE	Flash total size	Yes
FLASH_BASE_ADDRESS	Flash base memory address	if bl2 is built
FLASH_AREA_BL2_OFFSET	BL2 offset in flash	if bl2 is built
FLASH_AREA_BL2_SIZE	BL2 flash size	if bl2 is built
FLASH_PS_AREA_SIZE	Allocated size for the protected storage data in flash	Yes
FLASH_ITS_AREA_SIZE	Allocated size for the internal trusted storage data in flash	Yes
SECURE_IMAGE_OFFSET	Offset of the secure image data in flash	if bl2 is built
FLASH_DEV_NAME	Name as defined in the CMSIS flash drivers	Yes
TFM_HAL_PS_FLASH_DRIVER	Name as defined in the CMSIS flash drivers	used by protected storage partition
TFM_HAL_PS_SECTORS_PER_BLOCK	Number of physical erase sectors per logical FS block	used by protected storage partition
TFM_HAL_PS_PROGRAM_UNIT	Smallest flash programmable unit in bytes	used by protected storage partition
TFM_HAL_ITS_FLASH_DRIVER	Name as defined in the CMSIS flash drivers	used by internal trusted storage partition
TFM_HAL_ITS_SECTORS_PER_BLOCK	Number of physical erase sectors per logical ITS block	used by internal trusted storage partition
TFM_HAL_ITS_PROGRAM_UNIT	Smallest flash programmable unit in bytes	used by internal trusted storage partition
TFM_NV_COUNTERS_AREA_SIZE	Allocated size for the NV counters data in flash	if using TF-M templates

region_defs.h:

(MANDATORY)

This file can be anywhere in the platform folder, usually in a sub folder named `partition`. TF-M doesn't provide a template for this file, common practice is to copy it from another platform (e.g. arm/mps2/an521) and update the following entries.

General advice: if you don't know beforehand the size you will want for these elements you will have to make it iterative from an abitrary value taken from another platform (e.g. arm/mps2/an521)

Note: all size are in bytes

name	description	Requisiteness
BL2_HEAP_SIZE	Size of the Bootloader (MCUboot) heap	if bl2 is built
BL2_MSP_STACK_SIZE	(if bl2 is built) Size of the Bootloader (MCUboot) Main stack	if bl2 is built
S_HEAP_SIZE	Size of the Secure (S) world Heap	yes
S_MSP_STACK_SIZE	Size of the Secure (S) world Main stack	yes
S_PSP_STACK_SIZE	Size of the Secure (S) world Process stack	no for IPC model
NS_HEAP_SIZE	Size of the Non-Secure (NS) world Heap	if tfm_ns is built
NS_STACK_SIZE	Size of the Non-Secure (NS) world stack	if tfm_ns is built
PSA_INITIAL_ATTEST_MAX_TOKEN_SIZE	Size of the buffer that will store the initial attestation	used by initial attestation partition
TFM_ATTEST_BOOT_RECORDS_MAX_SIZE	Size of buffer that can store the encoded list of boot records	used by delegated attestation partition
BL2_HEADER_SIZE	Size of the Header for the Bootloader (MCUboot)	if bl2 is built
BL2_TRAILER_SIZE	Size of the Trailer for the Bootloader (MCUboot)	if bl2 is built
SHARED_SYMBOL_AREA_SIZE	Size of shared common code between bl2 and tfm_s	if bl2 is built and want to reduce image size

(OPTIONAL)

If the TF-M common linker script is used then:

name	description	Requisiteness
S_CODE_START	Start address for the S code	Yes
S_CODE_SIZE	Size of the S code	Yes
S_DATA_START	Start address for the S data	Yes
S_DATA_SIZE	Size of the S data	Yes
S_RAM_CODE_START	Start address for the S code	if no XIP on flash
S_RAM_CODE_SIZE	Size of the S code	if no XIP on flash

CMSIS_Driver/Config/cmsis_driver_config.h:

(location as defined in CMakeLists.txt)

This file should include the CMSIS drivers implementation headers.

CMSIS_Driver/Config/RTE_Device.h:

(location as defined in CMakeLists.txt)

This is the Run-Time Environnement file from CMSIS, which is there to allow enabling or disabling drivers prior to building. If your platform is designed as a general use platform, this file should contain all the available CMSIS drivers, and you should provide a recommended configuration. If your platform is designed for a specific use-case then you should reference and enable only the mandatory drivers.

CMSIS_Driver/Driver_Flash.c:

(location as defined in CMakeLists.txt)

TF-M relies on CMSIS Drivers, as such it requires the CMSIS functions to be implemented. As a platform owner you can decide to either implement the drivers in the CMSIS functions or to use the CMSIS functions as a shim to your native drivers.

Refer to the CMSIS FLASH documentation.

CMSIS_Driver/Driver_USART.c:

(location as defined in CMakeLists.txt)

TF-M relies on CMSIS Drivers, as such it requires the CMSIS functions to be implemented. As a platform owner you can decide to either implement the drivers in the CMSIS functions or to use the CMSIS functions as a shim to your native drivers.

Refer to the CMSIS USART documentation.

target_cfg.[ch]:

(location as defined in CMakeLists.txt)

It is expected that these files contain all platform specific code related to memory protection (e.g. SAU/PPC/MPC). These functions will not be called by TF-M directly, but are expected to be called from the function tfm_hal_set_up_static_boundaries() in tfm_hal_isolation.c.

tfm_hal_platform.c:

(location as defined in CMakeLists.txt)

Each platform is expected to implement the following API declared in platform/include/tfm_hal_platform.h

```
enum tfm_hal_status_t tfm_hal_platform_init(void);

The function will be called before SPM initialization.
```

tfm_hal_isolation.c:

(location as defined in CMakeLists.txt)

Each platform is expected to implement all the functions declared in platform/include/tfm_hal_isolation.h.

A reference implementation for Armv8-M platforms is provided in platform/ext/common/tfm_hal_isolation_v8m.c. Platforms using the common TF-M linker scripts and scatter files can use it to implement standard TF-M isolation with Armv8-M MPU regions. Platform-specific MPU regions can be appended by defining PLATFORM_STATIC_MPU_REGIONS in the platform's tfm_peripherals_def.h header.

These functions will be called from TF-M.

tfm_platform_system.c:

(location as defined in CMakeLists.txt)

Each platform is expected to implement all the functions declared in platform/include/tfm_platform_system.h.

check_config.cmake:

As a platform owner you may want to enforce some configuration or to prevent the use of unsupported configurations.

This file (CMake format) allows you to do so by allowing you to check for invalid configuration values.

This file is optional.

TF-M build system already provides a generic configuration checker that will be called on top of one provided by the platform owner. The generic checker is located in <TF-M ROOT>/config/.

[*check_config.cmake*]

platform_svc_numbers.h

(OPTIONAL)

If your platform has its own SVC handling, then you need to

- create the `platform_svc_numbers.h` which defines the platform SVC numbers.

  The bit [7] of the number must be set to 1 to reflect that it is a platform SVC number. The bit [6] indicates whether this SVC should be called from Handler mode or Thread mode. For more details of the bit assignments, please check the `svc_num.h`. TF-M provides two Macros `TFM_SVC_NUM_PLATFORM_THREAD(index)` and `TFM_SVC_NUM_PLATFORM_HANDLER(index)` to easily construct a valid number.

- implement the *platform_svc_handlers* function which handles SVC.
- enable `PLATFORM_SVC_HANDLERS` config option.

ns/CMakeLists.txt

(MANDATORY)

This is CMake script for building the platform support on NSPE side. It's copied to `<Artifact folder>` in the installation phase and instructs on how to build **platform_ns** target. The default NSPE build script expects this target definition and extends it with files, common for all TF-M platforms.

Note::

This file shall define and use paths of installed directories in `<Artifact folder>`, instead of paths in TF-M platform folder.

[*NSCMakeLists.txt*]

ns/cpuarch_ns.cmake

(MANDATORY for multi-core platforms)

This file contains the hardware information for the NSPE CPU. It should be installed to `<Artifact folder>/platform` for NSPE build, ranamed to `cpuarch.cmake`.

[*cpuarch.cmake*]

Functions

There are a few functions that need to be declared and properly initialized for TF-M to work. The function declarations can be found in platform/include/tfm_platform_system.h and platform/include/tfm_spm_hal.h.

tfm_platform_hal_system_reset:

This function will in most cases end up calling the NVIC System Reset.

The platform can uninitialize or store some resources before reset.

```
void tfm_platform_hal_system_reset(void);
```

tfm_platform_hal_ioctl:

A single entry point to platform-specific code across the HAL is provided by the IOCTL service.

```
enum tfm_platform_err_t tfm_platform_hal_ioctl(tfm_platform_ioctl_req_t request, psa_
↪invec  *in_vec, psa_outvec *out_vec);
```

tfm_hal_get_mem_security_attr:

Required on multi-core platforms only. This function shall fill the security_attr_info_t argument with the current active security configuration.

```
void tfm_hal_get_mem_security_attr(const void *p, size_t s, struct security_attr_info_t␣
↪*p_attr);
```

tfm_hal_get_secure_access_attr:

Required on multi-core platforms only. This function shall fill the mem_attr_info_t argument with the current active memory configuration of the target S memory region.

```
void tfm_hal_get_secure_access_attr(const void *p, size_t s, struct mem_attr_info_t *p_
↪attr);
```

tfm_hal_get_ns_access_attr:

Required on multi-core platforms only. This function shall fill the mem_attr_info_t argument with the current active memory configuration for the target NS memory region.

```
void tfm_hal_get_ns_access_attr(const void *p, size_t s, struct mem_attr_info_t *p_attr);
```

tfm_hal_irq_clear_pending:

This function clears any pending IRQ.

```
void tfm_hal_irq_clear_pending(uint32_t irq_num);
```

tfm_hal_irq_enable:

This function enable an IRQ.

```
void tfm_hal_irq_enable(uint32_t irq_num);
```

tfm_hal_irq_disable:

This function disable an IRQ.

```
void tfm_hal_irq_disable(uint32_t irq_num);
```

platform_svc_handlers

This function is the platform's SVC handler. It should return the result for callers and the SPM will then return it to the caller.

```
int32_t platform_svc_handlers(uint8_t svc_num, uint32_t *svc_args, uint32_t exc_return);
```

Annex

CMake build system snippets examples

CMakeLists.txt: Defining regions for Secure world platform and all linked to it.

```
target_include_directories(platform_region_defs
    INTERFACE
    <folder name under the platform folder - usually named platform>
)
```

CMakeLists.txt: Scatter files for SPE platform and bootloader

```
target_add_scatter_file(bl2
    $<$<C_COMPILER_ID:ARMClang>:${PLATFORM_DIR}/ext/common/armclang/tfm_common_bl2.sct>
    $<$<C_COMPILER_ID:GNU>:${PLATFORM_DIR}/ext/common/gcc/tfm_common_bl2.ld>
    $<$<C_COMPILER_ID:IAR>:${PLATFORM_DIR}/ext/common/iar/tfm_common_bl2.icf>
)
target_add_scatter_file(tfm_s
    $<$<C_COMPILER_ID:ARMClang>:${PLATFORM_DIR}/ext/common/armclang/tfm_common_s.sct>
    $<$<C_COMPILER_ID:GNU>:${PLATFORM_DIR}/ext/common/gcc/tfm_common_s.ld>
    $<$<C_COMPILER_ID:IAR>:${PLATFORM_DIR}/ext/common/iar/tfm_common_s.icf>
)
```

CMakeLists.txt: Startup files for SPE platform and bootloader

```
target_sources(bl2
    PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/platform/ext/target/<folder to platform>/device/source/
↪startup_<platform name>.c
)
target_sources(tfm_s
    PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/platform/ext/target/<folder to platform>/device/source/
↪startup_<platform name>.c
)
```

CMakeLists.txt: The Secure world platform sources

```
target_include_directories(platform_bl2
    PUBLIC
)
target_include_directories(platform_s
    PUBLIC
)

```

(continues on next page)

(continued from previous page)

```
target_sources(platform_bl2
    PRIVATE
)
target_sources(platform_s
    PRIVATE
)
target_sources(tfm_spm
    PRIVATE
        target_cfg.c
        tfm_hal_isolation.c
        tfm_hal_platform.c
)
```

CMakeLists.txt: installation for the Non-Secure world platform build

```
install(FILES ${PLATFORM_DIR}/ext/common/uart_stdout.c
              native_drivers/arm_uart_drv.c
              native_drivers/timer_cmsdk/timer_cmsdk.c
              cmsis_drivers/Driver_USART.c
              retarget/platform_retarget_dev.c
              cmsis_core/an521_ns_init.c
        DESTINATION ${INSTALL_PLATFORM_NS_DIR})

install(DIRECTORY ${PLATFORM_DIR}/ext/common
                  ${PLATFORM_DIR}/ext/driver
        DESTINATION ${INSTALL_PLATFORM_NS_DIR}/ext)
```

config.cmake

```
set(CONFIG_TFM_USE_TRUSTZONE            ON)
set(TFM_MULTI_CORE_TOPOLOGY             OFF)
set(BL2                                 OFF         CACHE BOOL      "Whether to build BL2
↪")
set(NS                                  FALSE       CACHE BOOL      "Whether to build NS␣
↪app" FORCE)
```

check_config.cmake

```
function(tfm_invalid_config)
    if (${ARGV})
        string (REPLACE ";" " " ARGV_STRING "${ARGV}")
        string (REPLACE "STREQUAL"     "=" ARGV_STRING "${ARGV_STRING}")
        string (REPLACE "GREATER"      ">" ARGV_STRING "${ARGV_STRING}")
        string (REPLACE "LESS"         "<" ARGV_STRING "${ARGV_STRING}")
        string (REPLACE "VERSION_LESS" "<" ARGV_STRING "${ARGV_STRING}")
        string (REPLACE "EQUAL"        "=" ARGV_STRING "${ARGV_STRING}")
        string (REPLACE "IN_LIST"      "in" ARGV_STRING "${ARGV_STRING}")

        message(FATAL_ERROR "INVALID CONFIG: ${ARGV_STRING}")
    endif()
endfunction()
```

(continues on next page)

(continued from previous page)

```
# Requires armclang >= 6.10.1
tfm_invalid_config((CMAKE_C_COMPILER_ID STREQUAL "ARMClang") AND (CMAKE_C_COMPILER_
↪VERSION VERSION_LESS "6.10.1"))
```

/ns/CMakeLists.txt:

```
add_library(platform_ns)

target_sources(platform_ns
    PRIVATE
        arm_uart_drv.c
        timer_cmsdk.c
        uart_stdout.c
        Driver_USART.c
    PUBLIC
        cmsis_core/startup_an521.c
)

target_include_directories(platform_ns
    PUBLIC
        include
        cmsis
        cmsis_core
)

target_compile_definitions(platform_ns
    PUBLIC
        $<$<BOOL:${PLATFORM_DEFAULT_CRYPTO_KEYS}>:PLATFORM_DEFAULT_CRYPTO_KEYS>
)
```

10.8.2 Platform Documentation

This document describes conventions for platform documentation for easy and smooth integration into TF-M online documentation.

Principles

- Minimise TF-M requirements and maximise flexibility to let platform vendors reuse existing documentation.
- Platforms are grouped under vendors' name. There is space to keep general information not specific to any of the vendors' platforms.
- A Table Of Contents (TOC) structures platforms' documentation. There is no defined template for platform documents.

Rules

- Platforms documentation is gathered in */docs/platform* folder. This folder is similar to the platforms implementation folder */platform/ext/target* but does not need to mirror its structure.
- A vendor is represented by a subfolder with *index.rst* as an entry point. This file may contain general materials about a vendor not specific to a particular platform.
- A vendor's *index.rst* shall carry a TOCtree structure with a list of all supported platforms at the top level.
- An explicit platform name is preferred in a vendor's TOCtree instead of use of platform's title by default.

Note: Do not forget to update the summary file after adding a new platform */docs/platform/platform_introduction.rst* included into TF-M introduction.

Example

Assuming a vendor *ABCD123* wants to add and document a new platform *Secure 1024* in TF-M. Here are the basic steps:

1. Create a folder */docs/platform/ABCD123/*
2. Create a platform documentation under */docs/platform/ABCD123*. A natural choice is to put a new platform under */docs/platform/ABCD123/Secure_1024* folder but it's not required.
3. Create a file */docs/platform/ABCD123/index.rst* with a platform entry, presuming it's location in */docs/platform/ABCD123/Secure_1024/index.rst*:

```
#################
ABCD123 platforms
#################

.. toctree::
Secure 1024 <Secure_1024/index>
```

4. Finally, update */docs/platform/index.rst* with a new vendor name:

```
##############
TF-M Platforms
##############

.. toctree::
    :maxdepth: 2
    :glob:

    Arm <arm/index.rst>
    NXP <nxp/index.rst>
    Cypress <cypress/index.rst>
    ...
    ABCD123 <ABCD123/index.rst>
```

10.8.3 Platform deprecation and removal

Process for Platform deprecation and removal

A platform may need to be removed from upstream code due to lack of community interest or it may have reached end of life. The below section calls out the process for removing platform support from TF-M.

1. An email to the TF-M mailing list proposing the removal of the platform.
2. The merit of the proposal will be considered by the maintainers for a period of 4 weeks and community can express their opinion on the same during this time window. The platform owner can veto the proposal and incase of multiple platform owners with differing opinion or community having interest in the platform, then the project maintainer can work with the platform owner and use their discretion to decide on the proposal.
3. Once a decision is made to remove the platform, the platform is considered to be in *deprecated* state as per platform support lifecyle defined here: "https://developer.trustedfirmware.org/w/collaboration/project-maintenance-process/". The platform will be marked as deprecated and the TF-M version after which it will be removed will also be mentioned. Suitable build time or runtime messages needs to be incorporated to the platform to warn about the *deprecation*.
4. The project should strive to keep the *deprecated* platforms building/running till the removal. This relies on platform owners being still actively involved with the project and maintaining the platform.
5. Although this will be the usual process for platform deprecation and eventual removal, the process still leaves room for the platform deprecation to be cancelled after it has been marked as *deprecated* due to evolving project and market requirements. This is left to consensus between project maintainers and platform owner/s.
6. Once a platform has been removed, it can be added back in future and this would follow the same guidelines as for a new platform contribution.

List of Deprecated Platforms

The below list calls out platforms marked for deprecation according to the above process and the platform will be removed soon after the mentioned release.

Deprecated Platform	Removed after release	Comments
arm/mps2/an539	v1.2.0	N.A
arm/mps2/sse-200_aws	v1.3.0	N.A
arm/musca_a	v1.3.0	N.A
arm/mps2/fvp_sse300	v1.4.0	N.A
arm/musca_b1/secure_enclave	v1.6.0	N.A

10.9 Services

10.9.1 Initial Attestation Service Integration Guide

Introduction

TF-M Initial Attestation Service allows the application to prove the device identity during an authentication process to a verification entity. The initial attestation service can create a token on request, which contains a fix set of device specific data.

TF-M Initial Attestation Service by default enables asymmetric key algorithm based attestation (*asymmetric attestation* for short). Symmetric key algorithm based attestation (*symmetric attestation* for short) can be enabled instead by selecting build option `SYMMETRIC_INITIAL_ATTESTATION`.

- In asymmetric attestation, device must contain an attestation key pair, which is unique per device. The token is signed with the private part of attestation key pair. The public part of the key pair is known by the verification entity. The public key is used to verify the token authenticity.
- In symmetric attestation, device should contain a symmetric attestation key to generate the authentication tag of token content. The verification entity uses the same symmetric key to verify the token authenticity.

The data items in the token used to verify the device integrity and assess its trustworthiness. Attestation key provisioning is out of scope for the attestation service and is expected to take part during manufacturing of the device.

Claims in the initial attestation token

The initial attestation token is formed of claims. A claim is a data item, which is represented in a key - value structure. The following fixed set of claims are included in the token:

- **Auth challenge**: Input object from caller. Can be a single nonce from server or hash of nonce and attested data. It is intended to provide freshness to report and the caller has responsibility to arrange this. Allowed length: 32, 48, 64 bytes. The claim is modeled to be eventually represented by the EAT standard claim nonce. Until such a time as that standard exists, the claim will be represented by a custom claim. Value is encoded as byte string.
- **Instance ID**: It represents the unique identifier of the instance. In the PSA definition it is:
  - a hash of the public attestation key of the instance in asymmetric attestation.
  - hashes of the symmetric attestation key of the instance in symmetric attestation.

  The claim is modeled to be eventually represented by the EAT standard claim UEID of type GUID. Until such a time as that standard exists, the claim will be represented by a custom claim Value is encoded as byte string.
- **Verification service indicator**: Optional, recommended claim. It is used by a Relying Party to locate a validation service for the token. The value is a text string that can be used to locate the service or a URL specifying the address of the service. The claim is modelled to be eventually represented by the EAT standard claim origination. Until such a time as that standard exists, the claim will be represented by a custom claim. Value is encoded as text string.
- **Profile definition**: Optional, recommended claim. It contains the name of a document that describes the 'profile' of the token, being a full description of the claims, their usage, verification and token signing. The document name may include versioning. Custom claim with a value encoded as text string.
- **Implementation ID**: Uniquely identifies the underlying immutable PSA RoT. A verification service can use this claim to locate the details of the verification process. Such details include the implementation's origin and associated certification state. Custom claim with a value encoded as byte string.

- **Client ID**: The partition ID of that secure partition or non-secure thread who called the initial attestation API. Custom claim with a value encoded as a *signed* integer. Negative number represents non-secure caller, positive numbers represents secure callers, zero is invalid.
- **Security lifecycle**: It represents the current lifecycle state of the instance. Custom claim with a value encoded as an integer.
- **Hardware version**: Optional claim. Globally unique number in EAN-13 format identifying the GDSII that went to fabrication, HW and ROM. It can be used to reference the security level of the PSA-ROT via a certification website. Custom claim with a value is encoded as text string.
- **Boot seed**: It represents a random value created at system boot time that will allow differentiation of reports from different system sessions. The size is 32 bytes. Custom claim with a value is encoded as byte string.
- **Software components**: Optional, but required in order to be compliant with the PSA-SM. It represents the software state of the system. The value of the claim is an array of CBOR map entries, with one entry per software component within the device. Each map contains multiple claims that describe evidence about the details of the software component.
- **No software measurements**: Optional, but required if no software component claims are made. In the event that the implementation does not contain any software measurements then it is mandatory to include this claim to indicate this is a deliberate state. Custom claim with a value encoded as an unsigned integer set to 1.

Each software component claim can include the following properties. Any property that is not optional must be included:

- **Measurement type**: Optional claim. It represents the role of the software component. Value is encoded as short(!) text string.
- **Measurement value**: It represents a hash of the invariant software component in memory at start-up time. The value must be a cryptographic hash of 256 bits or stronger. Value is encoded as byte string.
- **Version**: Optional claim. It represents the issued software version. Value is encoded as text string.
- **Signer ID**: Optional claim, but required in order to be compliant with the PSA-SM. It represents the hash of a signing authority public key. Value is encoded as byte string.
- **Measurement description**: Optional claim. It represents the way in which the measurement value of the software component is computed. Value is encoded as text string containing an abbreviated description (name) of the measurement method.

Initial attestation token (IAT) data encoding

The initial attestation token is planned to be aligned with future version of Entity Attestation Token format. The token is encoded according to the CBOR format and signed according to COSE standard.

Code structure

The PSA interface for the Initial Attestation Service is located in `interface/include`. The only header to be included by applications that want to use functions from the PSA API is `psa/initial_attestation.h`.

The TF-M Initial Attestation Service source files are located in `secure_fw/partitions/initial_attestation`.

Service source files

- **CBOR library**
  - `lib/ext/qcbor` This library is used to create a proper CBOR token. It can be used on 32-bit and 64-bit machines. It was designed to suite constrained devices with low memory usage and without dynamic memory allocation. Its source code is fetched automatically during the build configuration step from an external repository: QCBOR library.
  - `<qcbor_src>/inc/qcbor/qcbor_encode.h`: Public API documentation of CBOR library (encoding).
  - `<qcbor_src>/inc/qcbor/qcbor_decode.h`: Public API documentation of CBOR library (decoding).
- **COSE library:**
  - `lib/ext/t_cose`: This library is used to sign a CBOR token and create the COSE header and signature around the initial attestation token. Only a subset of the COSE standard is implemented. The COSE_Sign1 and COSE_Mac0 (only available in TF-M fork) signature schemas are supported.
  - It is a fork of this external t_cose library.
  - `lib/ext/t_cose/src/t_cose_crypto.h`: Expose an API to bind `t_cose` library with available crypto library in the device.
  - `lib/ext/t_cose/crypto_adapters/t_cose_psa_crypto.c`: Implements the exposed API and ports `t_cose` to the PSA Crypto API.
- **Initial Attestation Service:**
  - `attest_core.c` : Implements core functionalities such as implementation of APIs, retrieval of claims and token creation.
  - `attest_token_encode.c`: Implements the token creation functions such as start and finish token creation and adding claims to the token.
  - `attest_asymmetric_key.c`: Calculate the Instance ID value based on asymmetric initial attestation key.
  - `tfm_attest.c`: Implements the SPM abstraction layer, and bind the attestation service to the SPM implementation in TF-M project.
  - `tfm_attest_req_mngr.c`: Includes the initialization entry of attestation service and handles attestation service requests in IPC model.
  - `attest_symmetric_key.c`: Calculate the Instance ID value based on symmetric initial attestation key.

Service interface definitions

- **Boot loader interface**: The attestation service might include data in the token about the distinct software components in the device. This data is provided by the boot loader and must be encoded in the TLV format, definition is described below in the boot loader interface paragraph. Possible claims in the boot status are describe above in the software components paragraph.
- **Hardware abstraction layer:**
  - Headers are located in `platform/include` folder.
  - `tfm_attest_hal.h`: Expose an API to get the following claims: security lifecycle, verification service indicator, profile definition.

  - `tfm_plat_boot_seed.h`: Expose an API to get the boot seed claim.
  - `tfm_plat_device_id.h`: Expose an API to get the following claims: implementation ID, hardware version.

- **SPM interface:**
  - `attestation.h`: Expose an API to bind attestation service to an SPM implementation.
- **PSA interface:**
  - `psa/initial_attestation.h`: Public API definition of initial attestation service.
- **Crypto interface:**
  - `t_cose_crypto.h`: Expose an API to bind the `t_cose` implementation to any cryptographic library.

PSA interface

The TF-M Initial Attestation Service exposes the following PSA interface:

```
psa_status_t
psa_initial_attest_get_token(const uint8_t *auth_challenge,
                             size_t         challenge_size,
                             uint8_t       *token_buf,
                             size_t         token_buf_size,
                             size_t        *token_size);

psa_status_t
psa_initial_attest_get_token_size(size_t challenge_size,
                                  size_t *token_size);
```

The caller must allocate a large enough buffer, where the token is going to be created by Initial Attestation Service. The size of the created token is highly dependent on the number of software components in the system and the provided attributes of these. The `psa_initial_attest_get_token_size()` function can be called to get the exact size of the created token.

System integrators might need to port these interfaces to a custom secure partition manager implementation (SPM). Implementations in TF-M project can be found here:

- `interface/src/tfm_attest_api.c`: interface implementation.

Secure Partition Manager (SPM) interface

The Initial Attestation Service defines the following interface towards the secure partition manager (SPM). System integrators **must** port this interface according to their SPM implementation.

```
enum psa_attest_err_t
attest_get_boot_data(uint8_t major_type, void *ptr, uint32_t len);

enum psa_attest_err_t
attest_get_caller_client_id(int32_t *caller_id);
```

- `attest_get_boot_data()`: Service can retrieve the relevant data from shared memory area between boot loader and runtime software. It might be the case that only SPM has direct access to the shared memory area, therefore this function can be used to copy the service related data from shared memory to a local memory buffer. In TF-M implementation this function must be called during service initialization phase, because the shared memory region is deliberately overlapping with secure main stack to spare some memory and reuse this area during execution. If boot loader is not available in the system to provide attributes of software components then this function must be implemented in a way that just initialize service's memory buffer to:

```
struct shared_data_tlv_header *tlv_header = (struct shared_data_tlv_header *)ptr;
tlv_header->tlv_magic   = 2016;
tlv_header->tlv_tot_len = sizeof(struct shared_data_tlv_header *tlv_header);
```

- `attest_get_caller_client_id()`: Retrieves the ID of the caller thread.
- `tfm_client.h`: Service relies on the following external definitions, which must be present or included in this header file:

```
typedef struct psa_invec {
    const void *base;
    size_t len;
} psa_invec;

typedef struct psa_outvec {
    void *base;
    size_t len;
} psa_outvec;
```

Hardware abstraction layer

The following API definitions are intended to retrieve the platform specific claims. System integrators **must** implement these interface according to their SoC and software design. Detailed definition of the claims are above in the claims in the initial attestation token paragraph.

- `tfm_attest_hal_get_security_lifecycle()`: Get the security lifecycle of the device.
- `tfm_attest_hal_get_verification_service()`: Get the verification service indicator for initial attestation.
- `tfm_attest_hal_get_profile_definition()`: Get the name of the profile definition document for initial attestation.
- `tfm_plat_get_boot_seed()`: Get the boot seed, which is a constant random number during a boot cycle.
- `tfm_plat_get_implementation_id`: Get the implementation ID of the device.
- `tfm_plat_get_cert_ref`: Get the hardware version of the device.

Boot loader interface

It is **recommended** to have a secure boot loader in the boot chain, which is capable of measuring the runtime firmware components (calculates the hash value of firmware images) and provide other attributes of these (version, type, etc). If the used boot loader is not capable of sharing these information with the runtime software then the `BOOT_DATA_AVAILABLE` compiler flag **must** be set to OFF (see *Related compile time options*).

The shared data between boot loader and runtime software is TLV encoded. The definition of TLV structure is described in `bl2/include/tfm_boot_status.h`. The shared data is stored in a well known location in secure internal memory and this is a contract between boot loader and runtime SW.

The structure of shared data must be the following:

- At the beginning there must be a header: `struct shared_data_tlv_header` This contains a magic number and a size field which covers the entire size of the shared data area including this header.

```
struct shared_data_tlv_header {
    uint16_t tlv_magic;
    uint16_t tlv_tot_len;
};
```

- The header is followed by the entries which are composed from an entry header structure: `struct shared_data_tlv_entry` and the data. In the entry header there is a type and a length field. The `tlv_type` field identifies the consumer of the entry in the runtime software and specify the subtype of that data item. The `tlv_len` field covers the length of the data (not including the size of the entry header).

  After the entry header structure comes the actual data.

```
struct shared_data_tlv_entry {
    uint16_t tlv_type;
    uint16_t tlv_len;
};
```

- Arbitrary number and size of data entry can be in the shared memory area.

The figure below gives of overview about the `tlv_type` field in the entry header. The `tlv_type` always composed from a major and minorbnumber. Major number identifies the addressee in runtime software, which the databentry is sent to. Minor number used to encode more info about the data entry. The actual definition of minor number could change per major number. In case of boot status data, which is going to be processed by initial attestation service the minor number is split further to two part: `sw_module` and `claim`. The `sw_module` identifies the SW component in the system which the data item belongs to and the `claim` part identifies the exact type of the data.

`tlv_type` description:

```
|------------------------------------------------ |
|                  tlv_type (16 bits)             |
|-------------------------------------------------|
|   tlv_major(4 bits)    |   tlv_minor(12 bits)     |
|-------------------------------------------------|
| MAJOR_IAS   | sw_module(6 bits) | claim(6 bits) |
|-------------------------------------------------|
| MAJOR_CORE  |          TBD                      |
|-------------------------------------------------|
```

Overall structure of shared data:

```
---------------------------------------------------------------
| Magic number(uint16_t) | Shared data total length(uint16_t) |
---------------------------------------------------------------
| Major_type(4 bits) | Minor_type(12 bits) | Length(uint16_t) |
---------------------------------------------------------------
|                         Raw data                            |
---------------------------------------------------------------
|                              .                              |
|                              .                              |
|                              .                              |
---------------------------------------------------------------
| Major_type(4 bits) | Minor_type(12 bits) | Length(uint16_t) |
```

(continues on next page)

(continued from previous page)

```
---------------------------------------------------------------
|                          Raw data                          |
---------------------------------------------------------------
```

Crypto interface

Asymmetric key algorithm based attestation

Device **must** contain an asymmetric key pair. The private part of it is used to sign the initial attestation token. Current implementation supports only the ECDSA P256 signature over SHA256. The public part of the key pair is used to create the key identifier (kid) in the unprotected part of the COSE header. The kid is used by verification entity to look up the corresponding public key to verify the signature in the token. The *t_cose* part of the initial attestation service implements the signature generation and kid creation. But the actual calculation of token's hash and signature is done by the Crypto service in the device. System integrators might need to re-implement the following functions if they want to use initial attestation service with a different cryptographic library than Crypto service:

- `t_cose_crypto_pub_key_sign()`: Calculates the signature over a hash value.
- `t_cose_crypto_get_ec_pub_key()`: Get the public key to create the key identifier.
- `t_cose_crypto_hash_start()`: Start a multipart hash operation.
- `t_cose_crypto_hash_update()`: Add a message fragment to a multipart hash operation.
- `t_cose_crypto_hash_finish()`:Finish the calculation of the hash of a message.

Interface needed by verification code:

- `t_cose_crypto_pub_key_verify()`: Verify the signature over a hash value.

Key handling

The provisioning of the initial attestation key is out of scope of the service and this document. It is assumed that device maker provisions the unique asymmetric key pair during the manufacturing process. Software integrators **must** make sure that `TFM_BUILTIN_KEY_SLOT_IAK` is available via the Crypto service, which will then be used by the Attestation partition to perform the required signing operations via the PSA crypto interface.

Symmetric key algorithm based attestation

Device **must** contain a symmetric key to generate the authentication tag of the initial attestation token. A key identifier (kid) can be encoded in the unprotected part of the COSE header. It helps verification entity look up the symmetric key to verify the authentication tag in the token.

The *t_cose* part of the initial attestation service implements the authentication tag generation. The authentication tag generation is done by the Crypto service. System integrators might need to re-implement the following functions if platforms provide a different cryptographic library than Crypto service:

- `t_cose_crypto_hmac_sign_setup()`: Set up a multi-part HMAC calculation operation.
- `t_cose_crypto_hmac_update()`: Add a message fragment to a multi-part HMAC operation.
- `t_cose_crypto_hmac_sign_finish()`: Finish the calculation of the HMAC of a message.

Interface needed by verification code:

- `t_cose_crypto_hmac_verify_setup()`: Set up a multi-part HMAC verification operation.
- `t_cose_crypto_hmac_verify_finish()`: Finish the verification of the HMAC of a message.

It also requires the same hash operations as listed in asymmetric key algorithm based initial attestation above, in attestation test cases.

Key handling

The provisioning of the initial attestation key is out of scope of the service and this document. It is assumed that device maker provisions the symmetric key during the manufacturing process. The following API is defined to retrieve the symmetric attestation key from platform layer. Software integrators **must** port this interface according to their SoC design and make sure that key is available by Crypto service:

- `tfm_plat_get_symmetric_iak()`: Get the symmetric initial attestation key raw data.
- `tfm_plat_get_symmetric_iak_id()`: Get the key identifier of the symmetric initial attestation key. The key identifier can be used as `kid` parameter in COSE header. Optional.

Note: Asymmetric initial attestation and symmetric initial attestation may share the same HAL APIs in future development.

Initial Attestation Service compile time options

There is a defined set of flags that can be used to compile in/out certain service features. The `CommonConfig.cmake` file sets the default values of those flags. The list of flags are:

- `ATTEST_INCLUDE_OPTIONAL_CLAIMS`: Include also the optional claims to the attestation token. Default value: ON in base configure and profile large while OFF in profile small and medium.
- `ATTEST_INCLUDE_COSE_KEY_ID`: COSE key-id is an optional field in the COSE unprotected header. Key-id is calculated and added to the COSE header based on the value of this flag. Default value: OFF.
- `ATTEST_CLAIM_VALUE_CHECK`: Check attestation claims against hard-coded values found in `platform/ext/common/template/attest_hal.c`. Default value is OFF. Set to ON in a platform's CMake file if the attest HAL is not yet properly ported to it.
- `SYMMETRIC_INITIAL_ATTESTATION`: Select symmetric initial attestation. Default value: OFF.
- `ATTEST_INCLUDE_TEST_CODE`: The initial attestation implementation is instrumented with additional test code. This is required in order to run some of the initial attestation regression tests. These tests are not required to be run by platform integrators, and are only meant to be used for development or modification of the initial attestation implementation. Enabling this option enables T_COSE_DISABLE_SHORT_CIRCUIT_SIGN which will short circuit the signing operation. Default value: OFF.
- `ATTEST_STACK_SIZE`- Defines the stack size of the Initial Attestation Partition. This value mainly depends on the build type(debug, release and minisizerel) and compiler.

Related compile time options

- BOOT_DATA_AVAILABLE: The boot data is expected to be present in the shared data area between the boot loader and the runtime firmware when it's ON. Otherwise, when it's OFF does not check the content of the shared data area but instead assumes that the TLV header is present and valid (the magic number is correct) and there are no data entries. Its default value depends on the BL2 flag.

Comparison of asymmetric and symmetric algorithm based token authentication

The symmetric key based authentication requires a more complex infrastructure for key management. Symmetric keys must be kept secret because they are sensitive asset, like the private key in case of asymmetric cryptographic algorithms. The main difference is that private keys are only stored on device, with proper hardware protection against external access, but symmetric keys must be known by both party (device and verifier), so they must also be stored in a central server of a relying party (who verifies the tokens). If keys are revealed then devices can be impersonated. If the database with the symmetric keys becomes compromised then all corresponding devices become untrusted. Since a centralised database of symmetric keys may need to be network connected, this can be considered to be a valuable target for attackers. The advantage of ECDSA based token authentication is that sensitive assets is only stored one place (in the device) and only one unique key per device. So if a device is compromised then only that single device become untrusted. In this case, the database of the relying party contains the corresponding public keys, which are not considered to be a confidential assets, so they can be shared with anybody. This shows the main advantage of asymmetric based authentication, because verification of attestation tokens can be done by a third party, such as cloud service providers (CSP). Thus Device Maker (DM) or Chip Maker (CM) does not need to operate such a service.

	Symmetric	Asymmetric
Authentication mode	HMAC over SHA256	ECDSA P256 over SHA256
Crypto key type in HW	Symmetric key	ECDSA private key (secp256r1)
Secrets are stored	Device and database	Device only
Verification database contains	Same symmetric key	Public keys
COSE authentication tag in the token	COSE_Mac0	COSE_Sign1
Verification entity	CM or DM, who provisioned the symmetric key	Can be anybody: third party provisioning service, cloud service provider, CM, DM

Verification

Regression test

The initial attestation token is verified by the attestation test suite in test/secure_fw/suites/attestation. The test suite is responsible for verifying the token signature and parsing the token to verify its encoding and the presence of the mandatory claims. This test suite can be executed on the device. It is part of the regression test suite. The test suite is configurable in the test/secure_fw/suites/attestation/attest_token_test_values.h header file. In this file there are two attributes for each claim which are configurable (more details in the header file):

- Requirements of presence: optional or mandatory
- Expected value: Value check can be disabled or expected value can be provided here.

For initial attestation tests, the built-in IAK is used. Initial attestation regression test verifies the IAT generated by initial attestation service with the exported public key.

iat-verifier

There is another possibility to verify the attestation token. This addresses the off-device testing when the token is already retrieved from the device and verification is done on the requester side. There is a Python script for this purpose in the tf-m-tools repo called iat-verifier. It does the same checking as the attestation test suite. The following steps describe how to simulate an off-device token verification on a host computer. It is described how to retrieve an initial attestation token when TF-M code is executed on FVP and how to use the iat-verifier script to check the token. This example assumes that user has license for DS-5 and FVP models:

- Build TF-M with any of the `ConfigRegression*.cmake` build configurations for MPS2 AN521 platform. More info in *tfm_build_instruction*.
- Lunch FVP model in DS-5. More info in *Run TF-M examples on Arm platforms*.
- Set a breakpoint in `test/secure_fw/suites/attestation/attest_token_test.c` in `decode_test_internal(..)` after the `token_main_alt(..)` returned, i.e. on line 859. Execute the code in the model until the breakpoint hits second time. At this point the console prints the test case name:
  - For asymmetric initial attestation, the console prints `ECDSA signature test of attest token`.
  - For symmetric initial attestation, the console prints `Symmetric key algorithm based Initial Attestation test`.
- At this point the token resides in the model memory and can be dumped to host computer.
- The ADDRESS and SIZE attributes of the initial attestation token is stored in the `completed_token` local variable. Their value can be extracted in the `(x)=Variables` debug window.
- Apply commands below in the `Commands` debug window to dump the token in binary format to the host computer:
  - For asymmetric initial attestation `dump memory <PATH>/iat_01.cbor <ADDRESS> +<SIZE>`
  - For symmetric initial attestation `dump memory <PATH>/iat_hmac_02.cbor <ADDRESS> +<SIZE>`
- Execute commands below on the host computer to verify the token:
  - For asymmetric initial attestation `check_iat -p -K -k platform/ext/common/template/tfm_initial_attestation_key.pem <PATH>/iat_01.cbor`
  - For symmetric initial attestation `check_iat -m mac -p -K -k platform/ext/common/template/tfm_symmetric_iak.key <PATH>/iat_hmac_02.cbor`
- Documentation of the iat-verifier can be found in the tf-m-tools-iat-verifer.

10.9.2 Crypto Service Integration Guide

Introduction

The TF-M Crypto service allows Non Secure world applications and Secure services to use cryptographic functionalities such as symmetric ciphering, message signing and verification, asymmetric encryption and decryption, cryptographic hashes, message authentication codes (MACs), key derivation and agreement, authenticated encryption with associated data (AEAD). It exposes the PSA Crypto APIs[1].

The secure service resides in the Crypto partition as a single entry point and is made of different components:

[1] PSA Crypto APIs: https://armmbed.github.io/mbed-crypto/html/

- An interface layer that exposes the PSA Crypto API to either NS or S entities and is implemented in `interface/src/tfm_crypto_api.c`. The interface is based on the Uniform Secure Service Signature and communicates with the Secure Partition Manager available in TF-M
- An init module `secure_fw/partitions/crypto/crypto_init.c` that implements functionalities requested by TF-M during the initialisation phase, and an API dispatcher that at runtime receives the requests from the interface and dispatches them to the component that processes that particular API request
- A set of components that process cryptographic API requests, each component dispatching to a subset of functionalities, i.e. AEAD, Asymmetric, Ciphering, Hashing, Key derivation, Key management, MACs, and Random Number Generation
- An alloc module `secure_fw/partitions/crypto/crypto_alloc.c` that manages the partition secure memory, storing multipart application contexts, input / outputs of the APIs being requested, inaccessible from NS or other secure partitions
- A library abstraction module `secure_fw/partitions/crypto/crypto_library.c` which is used to abstract the details of the cryptographic library used by the service *backend* to provide the actual implementation of the crypto functionalities. The backend library must expose the PSA Crypto APIs, and must provide support to encode *key ownership* into key identifiers. This is not standardized by the PSA Crypto APIs so it must be provided as an extension to the APIs. The backend library also needs to provide the subsystem for key storage and retrieval, and, in case, the interface to the cryptographic accelerator of the underlying platform, using the PSA cryptoprocessor driver interface specification[2]. For this reason, it must provide a mechanism to access platform *builtin* keys, and permanent key slots using the *TF-M Internal Trusted Storage (ITS) service*, if available.

Code structure

The PSA interfaces for the Crypto service are located in `interface/include`. The only header to be included by applications that want to use functions from the PSA API is `psa/crypto.h`. The TF-M Crypto service source files are located in `secure_fw/partitions/crypto`.

PSA interfaces

The TF-M Crypto service exposes the PSA interfaces detailed in the header `psa/crypto.h`. This header, in turn, includes several other headers which are not meant to be included directly by user applications. For a detailed description of the PSA API interface, please refer to the comments in the `psa/crypto.h` header itself.

Service source files

A brief description of what is implemented by each source file is as below:

- `crypto_cipher.c` : Dispatcher for symmetric crypto operations
- `crypto_hash.c` : Dispatcher for hash operations
- `crypto_mac.c` : Dispatcher for MAC operations
- `crypto_aead.c` : dispatcher for AEAD operations
- `crypto_key_derivation.c` : Dispatcher for key derivation and key agreement operations
- `crypto_key_management.c` : Dispatcher for key management operations towards the key slot management system provided by the backend library
- `crypto_rng.c` : Dispatcher for the random number generation requests

[2] PSA cryptoprocessor driver interface: https://github.com/Mbed-TLS/mbedtls/blob/development/docs/proposed/psa-driver-interface.md

- crypto_asymmetric.c : Dispatcher for message signature/verification and encryption/decryption using asymmetric crypto
- crypto_init.c : Init module for the service. The modules stores also the internal buffer used to allocate temporarily the IOVECs needed, which is not required in case of SFN model. The size of this buffer is controlled by the CRYPTO_IOVEC_BUFFER_SIZE config define
- crypto_library.c : Library abstractions to interface the dispatchers towards the underlying library providing *backend* crypto functions. Currently this only supports the mbed TLS library. In particular, the mbed TLS library requires to provide a static buffer to be used as heap for its internal allocation. The size of this buffer is controlled by the CRYPTO_ENGINE_BUF_SIZE config define
- crypto_alloc.c : Takes care of storing multipart operation contexts in a secure memory not visible outside of the crypto service. The CRYPTO_CONC_OPER_NUM config define determines how many concurrent contexts are supported at once. In a multipart operation, the client view of the contexts is much simpler (i.e. just an handle), and the Alloc module keeps track of the association between handles and contexts
- tfm_crypto_api.c : This module is contained in interface/src and implements the PSA Crypto API client interface exposed to both S/NS clients. This module allows a configuration option CONFIG_TFM_CRYPTO_API_RENAME to be set to 1 in case the NS environment or integrators want to rename the API symbols exported by the TF-M Crypto service. The renaming adds a default prefix, tfm_crypto__ to all functions. The prefix can be changed editing the interface file. This config option is for the NS environment or integration setup only, hence it is not accessible through the TF-M config
- tfm_mbedcrypto_alt.c : This module is specific to the mbed TLS[3] library integration and provides some alternative implementation of mbed TLS APIs that can be used when a optimised profile is chosen. Through the _ALT mechanism it is possible to replace at link time default implementations available in mbed TLS with the ones available in this file

  Note: The _ALT mechanism will be deprecated in future releases of the mbed TLS library

Considerations on service configuration

Crypto *backend* library configuration

The TF-M Crypto service relies on a cryptographic library to provide the functionalities specific by the PSA Crypto API spec and the PSA cryptoprocessor driver interface spec. At the moment, the only supported library is mbed TLS[Page 237, 3].

The configuration of the backend library is supplied using the TFM_MBEDCRYPTO_CONFIG_PATH and TFM_MBEDCRYPTO_PSA_CRYPTO_CONFIG_PATH config option that point to configuration headers following the legacy mbed TLS configuration scheme or the new PSA based configuration scheme.

Platforms can specify an extra config file by setting the TFM_MBEDCRYPTO_PLATFORM_EXTRA_CONFIG_PATH variable (which is a wrapper around the MBEDTLS_USER_CONFIG_FILE option). This is preferred for platform configuration over TFM_MBEDCRYPTO_CONFIG_PATH and TFM_MBEDCRYPTO_PSA_CRYPTO_CONFIG_PATH as it does not interfere with config changes due to TFM Profile.

Note: The default entropy source configured for mbed TLS is MBEDTLS_ENTROPY_NV_SEED with a unique seed. For production devices, an alternative hardware entropy source can be specified using the config option MBEDTLS_ENTROPY_HARDWARE_ALT

[3] mbed TLS library: https://www.trustedfirmware.org/projects/mbed-tls/

Note: Starting from mbed TLS 3.3.0, the Python package `jsonschema` must be available when building as it is required by the autogen framework for the driver integrations into the PSA Crypto core and driver wrapper modules

Crypto service build time options

- `CRYPTO_STACK_SIZE` : Defines the stack size of the Crypto Secure Partition. This value might depend on several parameters such as the build type, the compiler being used, the cryptographic functionalities that are enabled at build time
- `CRYPTO_<COMPONENT>_MODULE_ENABLED` : A series of defines, one per each `<COMPONENT>` that processes cryptographic operations, that are used to disable modules at build time. Each define corresponds to a component as described in *the components list*.

Crypto service *builtin* keys integration

A detailed description of how the service interacts with *builtin* keys is available in the `tfm_builtin_key_loader` *design document*.

Note: The crypto service integration with builtin keys relies on implementation details of mbed TLS that are not standardized in the spec and might change between releases due to ongoing work[4]

References

10.9.3 TF-M Internal Trusted Storage Service Integration Guide

Introduction

TF-M Internal Trusted Storage (ITS) service implements PSA Internal Trusted Storage APIs.

The service is backed by hardware isolation of the flash access domain and relies on hardware to isolate the flash area from access by the Non-secure Processing Environment, as well as the Application Root of Trust at higher levels of isolation.

The current ITS service design relies on hardware abstraction provided by TF-M. The ITS service provides a non-hierarchical storage model, as a filesystem, where all the assets are managed by a linearly indexed list of metadata.

The design addresses the following high level requirements as well:

- **Confidentiality** - Resistance to unauthorised accesses through hardware/software attacks. Assumed to be provided by the internal flash device, backed by hardware isolation.
- **Access Authentication** - Mechanism to establish requester’s identity (a non-secure entity, secure entity, or a remote server).

[4] Interface for platform keys: https://github.com/ARM-software/psa-crypto-api/issues/550

- **Integrity** - Resistance to tampering by attackers with physical access is assumed to be provided by the internal flash device itself, while resistance to tampering by Non-secure or App RoT attackers also requires hardware isolation.
- **Reliability** - Resistance to power failure scenarios and incomplete write cycles.
- **Configurability** - High level of configurability to scale up/down memory footprint to cater for a variety of devices with varying requirements.
- **Performance** - Optimized to be used for resource constrained devices with very small silicon footprint, the PPA (power, performance, area) should be optimal.

Current ITS Service Limitations

- **Fragmentation** - The current design does not support fragmentation, as an asset is stored in a contiguous space in a block. This means that the maximum asset size can only be up-to a block size. Each block can potentially store multiple assets. A delete operation implicitly moves all the assets towards the top of the block to avoid fragmentation within block. However, this may also result in unutilized space at the end of each block.
- **Non-hierarchical storage model** - The current design uses a non-hierarchical storage model, as a filesystem, where all the assets are managed by a linearly indexed list of metadata. This model locates the metadata in blocks which are always stored in the same flash location. That increases the number of writes in a specific flash location as every change in the storage area requires a metadata update.
- **Protection against physical storage medium failure** - Complete handling of inherent failures of storage mediums (e.g. bad blocks in a NAND based device) is not supported by the current design.
- **Lifecycle management** - Currently, it does not support any subscription based keys and certificates required in a secure lifecycle management. Hence, an asset's validity time-stamp can not be invalidated based on the system time.
- **Provisioning vs user/device data** - In the current design, all assets are treated in the same manner. In an alternative design, it may be required to create separate partitions for provisioning content and user/device generated content. This is to allow safe update of provisioning data during firmware updates without the need to wipe out the user/device generated data.

Code Structure

TF-M Internal Trusted Storage service code is located in `secure_fw/partitions/internal_trusted_storage/` and is divided as follows:

- Core files
- Flash filesystem interfaces
- Flash interfaces

The PSA ITS interfaces for the TF-M ITS service are located in `interface/include/psa`.

PSA Internal Trusted Storage Interfaces

The TF-M ITS service exposes the following mandatory PSA ITS interfaces version 1.0:

```
psa_status_t psa_its_set(psa_storage_uid_t uid, size_t data_length, const void *p_data,␣
↪psa_storage_create_flags_t create_flags);
psa_status_t psa_its_get(psa_storage_uid_t uid, size_t data_offset, size_t data_size,␣
↪void *p_data, size_t *p_data_length);
psa_status_t psa_its_get_info(psa_storage_uid_t uid, struct psa_storage_info_t *p_info);
psa_status_t psa_its_remove(psa_storage_uid_t uid);
```

These PSA ITS interfaces and TF-M ITS types are defined and documented in `interface/include/psa/storage_common.h`, `interface/include/psa/internal_trusted_storage.h`, and `interface/include/tfm_its_defs.h`

Core Files

- `tfm_its_req_mngr.c` - Contains the ITS request manager implementation which handles all requests which arrive to the service. This layer extracts the arguments from the input and output vectors, and it calls the internal trusted storage layer with the provided parameters.
- `tfm_internal_trusted_storage.c` - Contains the TF-M internal trusted storage API implementations which are the entry points to the ITS service. Constructs a filesystem configuration using information from the ITS HAL, allocates a filesystem context for ITS and makes appropriate FS calls. Also handles requests from the PS partition with a separate FS config and context.
- `its_utils.c` - Contains common and basic functionalities used across the ITS service code.

Flash Filesystem Interface

- `flash_fs/its_flash_fs.h` - Abstracts the flash filesystem operations used by the internal trusted storage service. The purpose of this abstraction is to have the ability to plug-in other filesystems or filesystem proxies (supplicant).
- `flash_fs/its_flash_fs.c` - Contains the `its_flash_fs` implementation for the required interfaces.
- `flash_fs/its_flash_fs_mblock.c` - Contains the metadata block manipulation functions required to implement the `its_flash_fs` interfaces in `flash_fs/its_flash_fs.c`.
- `flash_fs/its_flash_fs_dblock.c` - Contains the data block manipulation functions required to implement the `its_flash_fs` interfaces in `flash_fs/its_flash_fs.c`.

The system integrator **may** replace this implementation with its own flash filesystem implementation or filesystem proxy (supplicant).

Flash Interface

The ITS filesystem flash interface is defined by `struct its_flash_fs_ops_t` in `flash_fs/its_flash_fs.h`.

Implementations of the ITS filesystem flash interface for different types of storage can be found in the ``internal_trusted_storage/flash` directory.

- `flash/its_flash.h` - Helper header that includes the correct ITS flash interface implementation for the target, and abstracts the allocation of different flash device types.
- `flash/its_flash_nand.c` - Implements the ITS flash interface for a NAND flash device, on top of the CMSIS flash interface implemented by the target. This implementation writes entire block updates in one-shot, so the CMSIS flash implementation **must** be able to detect incomplete writes and return an error the next time the block is read.
- `flash/its_flash_nor.c` - Implements the ITS flash interface for a NOR flash device, on top of the CMSIS flash interface implemented by the target.
- `flash/its_flash_ram.c` - Implements the ITS flash interface for an emulated flash device using RAM, on top of the CMSIS flash interface implemented by the target.

The CMSIS flash interface **must** be implemented for each target based on its flash controller.

The ITS flash interface depends on target-specific definitions from `platform/ext/target/<TARGET_NAME>/partition/flash_layout.h`. Please see the *Internal Trusted Storage Service HAL* section for details.

ITS Service Integration Guide

This section describes mandatory (i.e. **must** implement) or optional (i.e. **may** implement) interfaces which the system integrator has to take in to account in order to integrate the internal trusted storage service in a new platform.

Flash Interface

For ITS service operations, a contiguous set of blocks must be earmarked for the internal trusted storage area. The design requires either 2 blocks, or any number of blocks greater than or equal to 4. Total number of blocks can not be 0, 1 or 3. This is a design choice limitation to provide power failure safe update operations.

Maximum Asset Size

An asset is stored in a contiguous space in a logical filesystem block. The maximum size of an asset can be up-to the size of the data block. Typically, each logical block corresponds to one physical flash erase sector (the smallest unit that can erased), but the `TFM_HAL_ITS_SECTORS_PER_BLOCK` configuration below allows a number of contiguous erase sectors to form one logical block.

Internal Trusted Storage Service HAL

The ITS service requires the platform to implement the ITS HAL, defined in `platform/include/tfm_hal_its.h`.

The following C definitions in the HAL are mandatory, and must be defined by the platform in a header named `flash_layout.h`:

- `TFM_HAL_ITS_FLASH_DRIVER` - Defines the identifier of the CMSIS Flash ARM_DRIVER_FLASH object to use for ITS. It must have been allocated by the platform and will be declared extern in the HAL header.
- `TFM_HAL_ITS_PROGRAM_UNIT` - Defines the size of the ITS flash device's physical program unit (the smallest unit of data that can be individually programmed to flash). It must be equal to `TFM_HAL_ITS_FLASH_DRIVER.GetInfo()->program_unit`, but made available at compile time so that filesystem structures can be statically sized. Valid values are powers of two between 1 and the flash sector size, inclusive.

The following C definitions in the HAL may optionally be defined by the platform in the `flash_layout.h` header:

- `TFM_HAL_ITS_FLASH_AREA_ADDR` - Defines the base address of the dedicated flash area for ITS.
- `TFM_HAL_ITS_FLASH_AREA_SIZE` - Defines the size of the dedicated flash area for ITS in bytes.
- `TFM_HAL_ITS_SECTORS_PER_BLOCK` - Defines the number of contiguous physical flash erase sectors that form a logical erase block in the filesystem. The typical value is 1, but it may be increased so that the maximum required asset size will fit in one logical block.

If any of the above definitions are not provided by the platform, then the `tfm_hal_its_fs_info()` HAL API must be implemented instead. This function is documented in `tfm_hal_its.h`.

The sectors reserved to be used for Internal Trusted Storage **must** be contiguous.

Internal Trusted Storage Service Optional Platform Definitions

The following optional platform definitions may be defined in `flash_layout.h`:

- `ITS_RAM_FS_SIZE` - Defines the size of the RAM FS buffer when using the RAM FS emulated flash implementation. The buffer must be at least as large as the area earmarked for the filesystem by the HAL.
- `ITS_FLASH_NAND_BUF_SIZE` - Defines the size of the write buffer when using the NAND flash implementation. The buffer must be at least as large as a logical filesystem block.
- `ITS_MAX_BLOCK_DATA_COPY` - Defines the buffer size used when copying data between blocks, in bytes. If not provided, defaults to 256. Increasing this value will increase the memory footprint of the service.

More information about the `flash_layout.h` content, not ITS related, is available in *Details for the platform/ext folder* along with other platform information.

ITS Service Build Definitions

The ITS service uses a set of C definitions to compile in/out certain features, as well as to configure certain service parameters. When using the TF-M build system, these definitions are controlled by build flags of the same name. The `config/config_base.cmake` file sets the default values of those flags, but they can be overwritten based on platform capabilities by setting them in `platform/ext/target/<TARGET_NAME>/config.cmake`. The list of ITS service build definitions is:

- `ITS_CREATE_FLASH_LAYOUT`- this flag indicates that it is required to create an ITS flash layout. If this flag is set, ITS service will generate an empty and valid ITS flash layout to store assets. It will erase all data located in the assigned ITS memory area before generating the ITS layout. This flag is required to be set if the ITS memory area is located in a non-persistent memory. This flag can be set if the ITS memory area is located in a persistent memory without a valid ITS flash layout in it. That is the case when it is the first time in the device life that the ITS service is executed.

- `ITS_VALIDATE_METADATA_FROM_FLASH`- this flag allows to enable/disable the validation mechanism to check the metadata store in flash every time the flash data is read from flash. This validation is required if the flash is not hardware protected against data corruption.
- `ITS_RAM_FS`- setting this flag to `ON` enables the use of RAM instead of the persistent storage device to store the FS in the Internal Trusted Storage service. This flag is `OFF` by default. The ITS regression tests write/erase storage multiple time, so enabling this flag can increase the life of flash memory when testing. If this flag is set to `ON`, ITS_RAM_FS_SIZE must also be provided. This specifies the size of the block of RAM to be used to simulate the flash.

  Note: If this flag is disabled when running the regression tests, then it is recommended that the persistent storage area is erased before running the tests to ensure that all tests can run to completion. The type of persistent storage area is platform specific (eFlash, MRAM, etc.) and it is described in corresponding flash_layout.h

- `ITS_MAX_ASSET_SIZE` - Defines the maximum asset size to be stored in the ITS area. This size is used to define the temporary buffers used by ITS to read/write the asset content from/to flash. The memory used by the temporary buffers is allocated statically as ITS does not use dynamic memory allocation.
- `ITS_NUM_ASSETS` - Defines the maximum number of assets to be stored in the ITS area. This number is used to dimension statically the filesystem metadata tables in RAM (fast access) and flash (persistent storage). The memory used by the filesystem metadata tables is allocated statically as ITS does not use dynamic memory allocation.
- `ITS_BUF_SIZE`- Defines the size of the partition's internal data transfer buffer. If not provided, then `ITS_MAX_ASSET_SIZE` is used to allow asset data to be copied between the client and the filesystem in one iteration. Reducing the buffer size will decrease the RAM usage of the partition at the expense of latency, as data will be copied in multiple iterations. *Note:* when data is copied in multiple iterations, the atomicity property of the filesystem is lost in the case of an asynchronous power failure.
- `ITS_STACK_SIZE`- Defines the stack size of the Internal Trusted Storage Secure Partition. This value mainly depends on the platform specific flash drivers, the build type (Debug, Release and MinSizeRel) and compiler.

10.9.4 Platform Service Integration Guide

Introduction

TF-M Platform service is a trusted service which allows secure partitions and non-secure applications to interact with some platform-specific components. There are a number of features which requires some interaction with platform-specific components which are at the same time essential for the security of the system. Therefore, those components need to be handled by a secure partition which is part of the trusted compute base.

These platform-specific components include system reset, power management, Debug, GPIO, etc.

TF-M Platform interfaces

The TF-M interfaces for the Platform service are located in `interface/include/`. The TF-M Platform service source files are located in `secure_fw/partitions/platform`.

TF-M Platform service

The Platform service interfaces and types are defined and documented in `interface/include/tfm_platform_api.h`

- `platform_sp.h/c` : These files define and implement functionalities related to the platform service
- `tfm_platform_api.c` : This file implements `tfm_platform_api.h` functions to be called from the secure partitions. This is the entry point when the secure partitions request an action to the Platform service (e.g system reset).

Platform HAL

The Platform Service relies on a platform-specific implementation to perform some functionalities. Mandatory functionality (e.g. system reset) that are required to be implemented for a platform to be supported by TF-M have their dedicated HAL API functions. Additional platform-specific services can be provided using the IOCTL function call.

For API specification, please check: `platform/include/tfm_platform_system.h`

An implementation is provided in all the supported platforms. Please, check `platform/ext/target/<SPECIFIC_TARGET_FOLDER>/services/src/tfm_platform_system.c` for examples.

The API **must** be implemented by the system integrators for their targets.

IOCTL

A single entry point to platform-specific code across the HAL is provided by the IOCTL service and HAL function:

```
enum tfm_platform_err_t tfm_platform_hal_ioctl(tfm_platform_ioctl_req_t request,
                                               psa_invec *in_vec,
                                               psa_outvec *out_vec);
```

A request type is provided by the client, with additional parameters contained in the optional `in_vec` parameter. An optional output buffer can be passed to the service in `out_vec`. An IOCTL request type not supported on a particular platform should return `TFM_PLATFORM_ERR_NOT_SUPPORTED`

Non-Volatile counters

The Platform Service provides an abstracted service for exposing the NV counters to secure partitions or non-secure callers. The following operations are supported:

- Increment a counter.
- Read a counter value to a preallocated buffer.

```
enum tfm_platform_err_t
tfm_platform_nv_counter_increment(uint32_t counter_id);

enum tfm_platform_err_t
```

(continues on next page)

(continued from previous page)

```
tfm_platform_nv_counter_read(uint32_t counter_id,
                             uint32_t size, uint8_t *val);
```

The range of counters id is defined in : `platform/include/tfm_plat_nv_counters.h`

For Level 2,3 isolation implementations, secure partitions in the Application Root of Trust, should have `TFM_PLATFORM_SERVICE` set as a dependency for access to the NV counter API.

Current Service Limitations

- **system reset** - The system reset functionality is only supported in isolation level 1. Currently the mechanism by which PSA-RoT services should run in privileged mode in level 3 is not in place due to an ongoing work in TF-M Core. So, the `NVIC_SystemReset` call performed by the service is expected to generate a memory fault when it tries to access the `SCB->AIRCR` register in level 3 isolation.

10.9.5 Protected Storage Service Integration Guide

Introduction

TF-M Protected Storage (PS) service implements PSA Protected Storage APIs.

The service is usually backed by hardware isolation of the flash access domain and, in the current version, relies on hardware to isolate the flash area from non-secure access. In absence of hardware isolation, the secrecy and integrity of data is still maintained.

The PS service implements an AES-GCM based AEAD encryption policy, as a reference, to protect data integrity and authenticity.

The PS reuses the non-hierarchical filesystem provided by the TF-M Internal Trusted Storage service to store encrypted, authenticated objects.

The design addresses the following high level requirements as well:

- **Confidentiality** - Resistance to unauthorised accesses through hardware/software attacks.
- **Access Authentication** - Mechanism to establish requester's identity (a non-secure entity, secure entity, or a remote server).
- **Integrity** - Resistant to tampering by either the normal users of a product, package, or system or others with physical access to it. If the content of the protected storage is changed maliciously, the service is able to detect it.
- **Reliability** - Resistant to power failure scenarios and incomplete write cycles.
- **Configurability** - High level configurability to scale up/down memory footprint to cater for a variety of devices with varying security requirements.
- **Performance** - Optimized to be used for resource constrained devices with very small silicon footprint, the PPA (power, performance, area) should be optimal.

Current PS Service Limitations

- **Asset size limitation** - An asset is stored in a contiguous space in a block/sector. Hence, the maximum asset size can be up-to the size of the data block/sector. Detailed information about the maximum asset size can be found in the section *Maximum asset size* below.
- **Fragmentation** - The current design does not support fragmentation, as an asset is stored in a contiguous space in a block. Each block can potentially store multiple assets. A delete operation implicitly moves all the assets towards the top of the block to avoid fragmentation within block. However, this may also result in unutilized space at the end of each block.
- **Non-hierarchical storage model** - The current design uses a non-hierarchical storage model, as a filesystem, where all the assets are managed by a linearly indexed list of metadata. This model locates the metadata in blocks which are always stored in the same flash location. That increases the number of writes in a specific flash location as every change in the storage area requires a metadata update.
- **PSA internal trusted storage API** - In the current design, the service does not use the PSA Internal Trusted Storage API to write the rollback protection values stored in the internal storage.
- **Protection against physical storage medium failure** - Complete handling of inherent failures of storage mediums (e.g. bad blocks in a NAND based device) is not supported by the current design.
- **Key diversification** - In a more robust design, each asset would be encrypted through a different key.
- **Lifecycle management** - Currently, it does not support any subscription based keys and certificates required in a secure lifecycle management. Hence, an asset's validity time-stamp can not be invalidated based on the system time.
- **Provisioning vs user/device data** - In the current design, all assets are treated in the same manner. In an alternative design, it may be required to create separate partitions for provisioning content and user/device generated content. This is to allow safe update of provisioning data during firmware updates without the need to wipe out the user/device generated data.

Code Structure

Protected storage service code is located in `secure_fw/partitions/protected_storage/` and is divided as follows:

- Core files
- Cryptographic interfaces
- Non-volatile (NV) counters interfaces

The PSA PS interfaces for PS service are located in `interface/include/psa`

PSA Protected Storage Interfaces

The PS service exposes the following mandatory PSA PS interfaces, version 1.0:

```
psa_status_t psa_ps_set(psa_storage_uid_t uid, size_t data_length, const void *p_data,␣
↪psa_storage_create_flags_t create_flags);
psa_status_t psa_ps_get(psa_storage_uid_t uid, size_t data_offset, size_t data_size,␣
↪void *p_data, size_t *p_data_length);
psa_status_t psa_ps_get_info(psa_storage_uid_t uid, struct psa_storage_info_t *p_info);
psa_status_t psa_ps_remove(psa_storage_uid_t uid);
uint32_t psa_ps_get_support(void);
```

For the moment, it does not support the extended version of those APIs.

These PSA PS interfaces and PS TF-M types are defined and documented in `interface/include/psa/protected_storage.h`, `interface/include/psa/storage_common.h` and `interface/include/tfm_ps_defs.h`

Core Files

- `tfm_ps_req_mngr.c` - Contains the PS request manager implementation which handles all requests which arrive to the service. This layer extracts the arguments from the input and output vectors, and it calls the protected storage layer with the provided parameters.
- `tfm_protected_storage.c` - Contains the TF-M protected storage API implementations which are the entry points to the PS service.
- `ps_object_system.c` - Contains the object system implementation to manage all objects in PS area.
- `ps_object_table.c` - Contains the object system table implementation which complements the object system to manage all object in the PS area. The object table has an entry for each object stored in the object system and keeps track of its version and owner.
- `ps_encrypted_object.c` - Contains an implementation to manipulate encrypted objects in the PS object system.
- `ps_utils.c` - Contains common and basic functionalities used across the PS service code.
- `ps_filesystem_interface.c` - Contains the interface to directly use ITS implementation of services.

Flash Filesystem and Flash Interfaces

The non-hierarchical filesystem and flash interfaces reside under Internal Trusted Storage service directory. The way PS service uses them depends on the config option `TFM_PARTITION_INTERNAL_TRUSTED_STORAGE`.

When TF-M Internal trusted storage service is active (i.e. `TFM_PARTITION_INTERNAL_TRUSTED_STORAGE` = ON), all file system and storage interfaces are built as part of ITS service library. Thus, the PS service stores encrypted, authenticated objects by making service calls to the ITS service.

When TF-M Internal trusted storage service is disabled (i.e. `TFM_PARTITION_INTERNAL_TRUSTED_STORAGE` = OFF), all file system and storage interfaces are built as part of PS service library. Thus, the PS service stores encrypted, authenticated objects by making standard function calls to the file system within its own partition code.

The ITS filesystem and flash interfaces and their implementation can be found in `secure_fw/partitions/internal_trusted_storage/flash_fs` and `secure_fw/partitions/internal_trusted_storage/flash` respectively. More information about the filesystem and flash interfaces can be found in the *ITS integration guide*.

The ITS service implementation in `secure_fw/partitions/internal_trusted_storage/tfm_internal_trusted_storage.c`, constructs a filesystem configuration for Protected Storage based on target-specific definitions from the Protected Storage HAL. Please see the *Protected Storage Service HAL* section for details of these.

Cryptographic Interface

- `crypto/ps_crypto_interface.h` - Abstracts the cryptographic operations for the protected storage service.
- `crypto/ps_crypto_interface.c` - Implements the PS service cryptographic operations with calls to the TF-M Crypto service.

Non-volatile (NV) Counters Interface

The current PS service provides rollback protection based on NV counters. PS defines and implements the following NV counters functionalities:

- `nv_counters/ps_nv_counters.h` - Abstracts PS non-volatile counters operations. This API detaches the use of NV counters from the TF-M NV counters implementation, provided by the platform, and provides a mechanism to compile in a different API implementation for test purposes. A PS test suite **may** provide its own custom implementation to be able to test different PS service use cases.
- `nv_counters/ps_nv_counters.c` - Implements the PS NV counters interfaces based on TF-M NV counters implementation provided by the platform.

PS Service Integration Guide

This section describes mandatory (i.e. **must** implement) or optional (i.e. **may** implement) interfaces which the system integrator have to take in to account in order to integrate the protected storage service in a new platform.

Maximum Asset Size

An asset is stored in a contiguous space in a block/sector. The maximum size of an asset can be up-to the size of the data block/sector minus the object header size (`PS_OBJECT_HEADER_SIZE`) which is defined in `ps_object_defs.h`. The `PS_OBJECT_HEADER_SIZE` changes based on the `PS_ENCRYPTION` flag status.

Protected Storage Service HAL

The PS service requires the platform to implement the PS HAL, defined in `platform/include/tfm_hal_ps.h`.

The following C definitions in the HAL are mandatory, and must be defined by the platform in a header named `flash_layout.h`:

- `TFM_HAL_PS_FLASH_DRIVER` - Defines the identifier of the CMSIS Flash ARM_DRIVER_FLASH object to use for PS. It must have been allocated by the platform and will be declared extern in the HAL header.
- `TFM_HAL_PS_PROGRAM_UNIT` - Defines the size of the PS flash device's physical program unit (the smallest unit of data that can be individually programmed to flash). It must be equal to `TFM_HAL_PS_FLASH_DRIVER.GetInfo()->program_unit`, but made available at compile time so that filesystem structures can be statically sized. Valid values are powers of two between 1 and the flash sector size, inclusive.

The following C definitions in the HAL may optionally be defined by the platform in the `flash_layout.h` header:

- `TFM_HAL_PS_FLASH_AREA_ADDR` - Defines the base address of the dedicated flash area for PS.
- `TFM_HAL_PS_FLASH_AREA_SIZE` - Defines the size of the dedicated flash area for PS in bytes.
- `TFM_HAL_PS_SECTORS_PER_BLOCK` - Defines the number of contiguous physical flash erase sectors that form a logical erase block in the filesystem. The typical value is 1, but it may be increased so that the maximum required asset size will fit in one logical block.

If any of the above definitions are not provided by the platform, then the `tfm_hal_ps_fs_info()` HAL API must be implemented instead. This function is documented in `tfm_hal_ps.h`.

The sectors reserved to be used for Protected Storage **must** be contiguous sectors starting at `TFM_HAL_PS_FLASH_AREA_ADDR`.

The design requires either 2 blocks, or any number of blocks greater than or equal to 4. Total number of blocks can not be 0, 1 or 3. This is a design choice limitation to provide power failure safe update operations.

Protected Storage Service Optional Platform Definitions

The following optional platform definitions may be defined in `flash_layout.h`:

- `PS_RAM_FS_SIZE` - Defines the size of the RAM FS buffer when using the RAM FS emulated flash implementation. The buffer must be at least as large as the area earmarked for the filesystem by the HAL.
- `PS_FLASH_NAND_BUF_SIZE` - Defines the size of the write buffer when using the NAND flash implementation. The buffer must be at least as large as a logical filesystem block.

More information about the `flash_layout.h` content, not ITS related, is available in *Details for the platform/ext folder* along with other platform information.

TF-M NV Counter Interface

To have a platform independent way to access the NV counters, TF-M defines a platform NV counter interface. For API specification, please check: `platform/include/tfm_plat_nv_counters.h`

The system integrators **may** implement this interface based on the target capabilities and set the `PS_ROLLBACK_PROTECTION` flag to compile in the rollback protection code.

> **Note:** If this flag is enabled, the lifecycle of the PS service depends on the shorter write endurance of the assets storage device and the NV counters storage device.

Secret Platform Unique Key

The encryption policy relies on a secret hardware unique key (HUK) per device. It is system integrator's responsibility to provide an implementation which **must** be a non-mutable target implementation. For API specification, please check: `platform/include/tfm_plat_crypto_keys.h`

A stub implementation is provided in `platform/ext/common/template/crypto_keys.c`

Non-Secure Identity Manager

TF-M core tracks the current client IDs running in the secure or non-secure processing environment. It provides a dedicated API to retrieve the client ID which performs the service request.

Non-secure Client Extension Integration Guide provides further details on how client identification works.

PS service uses that TF-M core API to retrieve the client ID and associate it as the owner of an asset. Only the owner can read, write or delete that asset based on the creation flags.

The *integration guide* provides further details of non-secure implementation requirements for TF-M.

Cryptographic Interface

The reference encryption policy is built on AES-GCM, and it **may** be replaced by a vendor specific implementation.

The PS service abstracts all the cryptographic requirements and specifies the required cryptographic interface in `secure_fw/partitions/protected_storage/crypto/ps_crypto_interface.h`

The PS service cryptographic operations are implemented in `secure_fw/partitions/protected_storage/crypto/ps_crypto_interface.c`, using calls to the TF-M Crypto service.

PS Service Build Definitions

The PS service uses a set of C definitions to compile in/out certain features, as well as to configure certain service parameters. When using the TF-M build system, these definitions are controlled by build flags of the same name. The `config/config_base.cmake` file sets the default values of those flags, but they can be overwritten based on platform capabilities by setting them in `platform/ext/target/<TARGET_NAME>/config.cmake`. The list of PS service build definitions is:

- `PS_ENCRYPTION`- this flag allows to enable/disable encryption option to encrypt the protected storage data.
- `PS_CRYPTO_AEAD_ALG` - this flag indicates the AEAD algorithm to use for authenticated encryption in Protected Storage.

  Note: For GCM/CCM it is essential that IV doesn't get repeated. If this flag is set to `PSA_ALG_GCM` or `PSA_ALG_CCM`, `PS_ROLLBACK_PROTECTION` must be enabled to protect against IV rollback.

- `PS_CREATE_FLASH_LAYOUT`- this flag indicates that it is required to create a PS flash layout. If this flag is set, PS service will generate an empty and valid PS flash layout to store assets. It will erase all data located in the assigned PS memory area before generating the PS layout. This flag is required to be set if the PS memory area is located in a non-persistent memory. This flag can be set if the PS memory area is located in a persistent memory without a valid PS flash layout in it. That is the case when it is the first time in the device life that the PS service is executed.
- `PS_VALIDATE_METADATA_FROM_FLASH`- this flag allows to enable/disable the validation mechanism to check the metadata store in flash every time the flash data is read from flash. This validation is required if the flash is not hardware protected against malicious writes. In case the flash is protected against malicious writes (i.e embedded flash, etc), this validation can be disabled in order to reduce the validation overhead.
- `PS_ROLLBACK_PROTECTION`- this flag allows to enable/disable rollback protection in protected storage service. This flag takes effect only if the target has non-volatile counters and `PS_ENCRYPTION` flag is on.
- `PS_RAM_FS`- setting this flag to `ON` enables the use of RAM instead of the persistent storage device to store the FS in the Protected Storage service. This flag is `OFF` by default. The PS regression tests write/erase storage multiple time, so enabling this flag can increase the life of flash memory when testing. If this flag is set to `ON`, PS_RAM_FS_SIZE must also be provided. This specifies the size of the block of RAM to be used to simulate the flash.

  Note: If this flag is disabled when running the regression tests, then it is recommended that the persistent storage area is erased before running the tests to ensure that all tests can run to completion. The type of persistent storage area is platform specific (eFlash, MRAM, etc.) and it is described in corresponding flash_layout.h

- `PS_MAX_ASSET_SIZE` - Defines the maximum asset size to be stored in the PS area. This size is used to define the temporary buffers used by PS to read/write the asset content from/to flash. The memory used by the temporary buffers is allocated statically as PS does not use dynamic memory allocation.

- `PS_NUM_ASSETS` - Defines the maximum number of assets to be stored in the PS area. This number is used to dimension statically the object table size in RAM (fast access) and flash (persistent storage). The memory used by the object table is allocated statically as PS does not use dynamic memory allocation.
- `PS_TEST_NV_COUNTERS`- this flag enables the virtual implementation of the PS NV counters interface in `test/secure_fw/suites/ps/secure/nv_counters` of the `tf-m-tests` repo, which emulates NV counters in RAM, and disables the hardware implementation of NV counters provided by the secure service. This flag is enabled by default, but has no effect when the secure regression test is disabled. This flag can be overridden to `OFF` when building the regression tests. In this case, the PS rollback protection test suite will not be built, as it relies on extra functionality provided by the virtual NV counters to simulate different rollback scenarios. The remainder of the PS test suites will run using the hardware NV counters. Please note that running the tests in this configuration will quickly increase the hardware NV counter values, which cannot be decreased again. Overriding this flag from its default value of `OFF` when not building the regression tests is not currently supported.
- `PS_STACK_SIZE`- Defines the stack size of the Protected Storage Secure Partition. This value mainly depends on the build type(debug, release and minisizerel) and compiler.

10.9.6 Adding Secure Partition

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 36:: term table

Term	Meaning
FF-M	Firmware Framework for M
ID	Identifier
IPC	Interprocess communication
IPC model	The secure IPC framework
irqs	Interrupt requests
MMIO	Memory Mapped I/O
PSA	Platform Security Architecture
RoT	Root of Trust
SFN	Secure Function
SFN model	Secure Function model
SID	RoT Service ID
SP	Secure Partition
SPM	Secure Partition Manager
TF-M	Trusted firmware M

Introduction

Secure Partition is an execution environment that provides the following functions to Root of Trust (RoT) Services:

- Access to resources, protection of its own code and data.
- Mechanisms to interact with other components in the system.

Each Secure Partition is a single thread of execution and the smallest unit of isolation.

This document mainly describes how to add a secure partition in TF-M and focuses on the configuration, manifest, implement rules. The actual source-level implementation is not included in this document.

Note: If not otherwise specified, the steps are identical for IPC and SFN model.

The IPC and SFN model conforms to the *PSA Firmware Framework for M (FF-M) v 1.1* changes. Refer to PSA Firmware Framework specification and Firmware Framework for M 1.1 Extensions for details.

Process

The main steps to add a secure partition are as follows:

- *Add source folder*
- *Add manifest*
- *Update the Build System*
- *Implement the RoT services*

Add source folder

Add a source folder under `<TF-M base folder>/secure_fw/partitions` for the new secure partition (Let's take `example` as the folder name):

This folder should include those parts:

- Manifest file
- CMake configuration files
- Source code files

Add manifest

Each Secure Partition must have resource requirements declared in a manifest file. The Secure Partition Manager (SPM) uses the manifest file to assemble and allocate resources within the SPE. The manifest includes the following:

- A Secure Partition name.
- A list of implemented RoT Services.
- Access to other RoT Services.
- Memory requirements.
- Scheduling hints.
- Peripheral memory-mapped I/O regions and interrupts.

Note: The current manifest format in TF-M is “yaml” which is different from the requirement of PSA FF.

Note: The users can use LOW, NORMAL and HIGH to determine the priority of the Secure Partition in manifest. They are replaced by 01, 02 and 03 auotmatically when parsing manifest lists for section naming.

Here is a manifest reference example for the IPC model:

Note: To use SFN model, the user needs to replace `"model": "IPC"` to `"model": "SFN"`. The user also needs to remove the attribute `"entry_point"`, and optionally replace it with `"entry_init"`.

```
{
  "psa_framework_version": 1.1,
  "name": "TFM_SP_EXAMPLE",
  "type": "APPLICATION-ROT",
  "priority": "NORMAL",
  "model": "IPC",
  "entry_point": "tfm_example_main",
  "stack_size": "0x0200",
  "services" : [
    {
      "name": "ROT_A",
      "sid": "0x000000E0",
      "non_secure_clients": true,
      "connection_based": true,
      "version": 1,
      "version_policy": "STRICT"
      "mm_iovec": "disable"
    }
  ],
  "mmio_regions": [
    {
      "name": "TFM_PERIPHERAL_A",
      "permission": "READ-WRITE"
    }
  ],
  "irqs": [
    {
      "source": "TFM_A_IRQ",
      "name": "A_IRQ",
      "handling": "SLIH"
    }
  ]
  "dependencies": [
    "TFM_CRYPTO",
    "TFM_INTERNAL_TRUSTED_STORAGE_SERVICE"
  ]
}
```

Update manifest list

The <TF-M base folder>/tools/tfm_manifest_list.yaml is used to collect necessary information of secure partition. The manifest tool tools/tfm_parse_manifest_list.py processes it and generates necessary files while building.

Please refer to the *Manifest List* for the format of manifest lists.

Reference configuration example:

```
{
  "description": "TFM Example Partition",
  "manifest": "secure_fw/partitions/example/tfm_example_partition.yaml",
  "conditional": "@TFM_PARTITION_EXAMPLE@",
  "output_path": "partitions/example",
  "version_major": 0,
  "version_minor": 1,
  "pid": 290,
  "linker_pattern": {
    "library_list": [
      "*tfm_*partition_example*"
     ]
  }
}
```

TF-M also supports out-of-tree Secure Partition build where you can have your own manifest lists. Please refer to *Out-of-tree Secure Partition build* for details.

Secure Partition ID Distribution

Every Secure Partition has an identifier (ID). TF-M will generate a header file that includes definitions of the Secure Partition IDs. The header file is <TF-M build folder>generated/interface/include/psa_manifest/pid.h. Each definition uses the name attribute in the manifest as its name and the value is allocated by SPM.

The Partition ID can be set to a fixed value or omitted to be auto allocated.

```
#define name id-value
```

Table 37:: PID table

Secure Partitions	PID Range
TF-M Internal Partitions	0 - 255
PSA and user Partitions	256 - 2999
TF-M test Partitions	3000 - 4999
Firmware Framework test Partitions	5000 - 5999
Reserved	6000 -

Please refer to <TF-M base folder>/tools/tfm_manifest_list.yaml, <TF-M test repo>/test/secure_fw/tfm_test_manifest_list.yaml and <TF-M base folder>/tools/tfm_psa_ff_test_manifest_list.yaml for the detailed PID allocations.

About where to add the definition, please refer to the chapter *Update manifest list*.

RoT Service ID (SID) Distribution

An RoT Service is identified by its RoT Service ID (SID). A SID is a 32-bit number that is associated with a symbolic name in the Secure Partition manifest. The bits [31:12] uniquely identify the vendor of the RoT Service. The remaining bits [11:0] can be used at the discretion of the vendor.

Here is the RoT Service ID table used in TF-M.

Table 38:: SID table

Partitions	Vendor ID(20 bits)	Function ID(12 bits)
initial_attestation	0x00000	0x020-0x03F
platform	0x00000	0x040-0x05F
protected_storage	0x00000	0x060-0x06F
internal_trusted_storage	0x00000	0x070-0x07F
crypto	0x00000	0x080-0x09F
firmware_update	0x00000	0x0A0-0x0BF
tfm_secure_client	0x0000F	0x000-0x01F
tfm_ipc_client	0x0000F	0x060-0x07F
tfm_ipc_service	0x0000F	0x080-0x09F
tfm_slih_test_service	0x0000F	0x0A0-0x0AF
tfm_flih_test_service	0x0000F	0x0B0-0x0BF
tfm_ps_test_service	0x0000F	0x0C0-0x0DF
tfm_secure_client_2	0x0000F	0x0E0-0x0FF
tfm_sfn_test_service_1	0x0000F	0x100-0x11F
tfm_sfn_test_service_2	0x0000F	0x120-0x13F
tfm_attest_test_service	0x0000F	0x140-0x15F

RoT Service Stateless Handle Distribution

A Secure partition may include stateless services. They are distinguished and referenced by stateless handles. In manifest, a `stateless_handle` attribute is set for indexing stateless services. It must be either `"auto"` or a number in the range [1, 32] in current implementation and may extend. Also the `connection-based` attribute must be set to `false` for stateless services.

The indexes of stateless handles are divided into two ranges for different usages. Indexes [1, 16] are assigned to TF-M Secure Partitions. The rest indexes [17, 32] are reserved for any other Secure Partitions, for example Secure Partitions in `tf-m-tests` and `tf-m-extras`.

The following table summaries the stateless handle allocation for the TF-M Secure Partitions.

Table 39:: Stateless Handle table

Partition name	Stateless Handle
TFM_SP_CRYPTO	1
TFM_SP_PS	2
TFM_SP_ITS	3
TFM_SP_INITIAL_ATTESTATION	4
TFM_SP_FWU	5
TFM_SP_PLATFORM	6

For the indexes of other Secure Partitions, please refer to their manifests or documentations.

stack_size

The stack_size is required to indicate the stack memory usage of the Secure Partition. The value of this attribute must be a decimal or hexadecimal value in bytes. It can also be a build configurable with default value defined in config_base.cmake. The value of the configuration can be overridden to fit different use cases.

heap_size

This attribute is optional. The default value is 0. It indicates the heap memory usage of the Secure Partition. The allowed values are the same as the stack_size.

mmio_regions

This attribute is a list of MMIO region objects which the Secure Partition needs access to. TF-M only supports the named_region current. Please refer to PSA FF for more details about it. The user needs to provide a name macro to indicate the variable of the memory region.

TF-M uses the below structure to indicate a peripheral memory.

```
struct platform_data_t {
  uint32_t periph_start;
  uint32_t periph_limit;
  int16_t periph_ppc_bank;
  int16_t periph_ppc_loc;
};
```

Note: This structure is not expected by TF-M, it's only that the current implementations are using. Other peripherals that need different information to create isolation need to define a different structure with the same name.

Here is an example for it:

```
struct platform_data_t tfm_peripheral_A;
#define TFM_PERIPHERAL_A                 (&tfm_peripheral_A)
```

mm_iovec

Memory-mapped iovecs (MM-IOVEC) provides direct mapping of client input and output vectors into the Secure Partition. When this attribute is set to enable, it enables Secure Partitions to use the MM-IOVEC APIs if the framework supports MM-IOVEC.

Using MM-IOVEC provides a memory and runtime optimization for larger buffers, but reduces mitigation for common security vulnerabilities. Please refer to Firmware Framework for M 1.1 Extensions for more details. Whether to use MM-IOVEC depends on the requirements of memory and runtime optimization and security.

Update the Build System

The following changes to the build system are required for the newly added secure partition.

Add a CMakeLists.txt file

Each Secure Partition must have a corresponding CMakeLists.txt, in this case, <TF-M base folder>/secure_fw/partitions/example/CMakeLists.txt, which is the compilation configuration for this Secure Partition.

Here is a reference example for CMakeLists.txt

The CMake file should include the following contents

- Add library tfm_app_rot_partition_example and associated source files.

```
add_library(tfm_app_rot_partition_example STATIC)

target_sources(tfm_app_rot_partition_example
    PRIVATE
        tfm_example_partition.c
)
```

Note: The secure partition must be built as a standalone static library, and the name of the library must follow this pattern, as it affects how the linker script will lay the partition in memory:

  - tfm_psa_rot_partition* in case of a PSA RoT partition
  - tfm_app_rot_partition* in case of an Application RoT partition

- Add source files generated by the manifest tool.

```
# The intermedia file defines the partition stack.
target_sources(tfm_app_rot_partition_example
    PRIVATE
        ${CMAKE_BINARY_DIR}/generated/example_partition/auto_generated/intermedia_
↪tfm_example_partition.c
)

# The load info file includes the static data of the partition.
target_sources(tfm_partitions
    INTERFACE
        ${CMAKE_BINARY_DIR}/generated/example_partition/auto_generated/load_info_
↪tfm_example_partition.c
)
```

- Add dependency with manifest tool.

  To make sure the above generated files are up-to-date when the Secure Partition library is built, dependencies between the library and the manifest tool target should be set up.

```
add_dependencies(tfm_app_rot_partition_example manifest_tool)
```

- Link tfm_sprt for the PSA API interfaces.

```
target_link_libraries(tfm_app_rot_partition_example
    PRIVATE
        tfm_sprt
)
```

- Link the Secure Partition library to `tfm_partitions` so that it can be included in the final image.

```
target_link_libraries(tfm_partitions
    INTERFACE
        tfm_app_rot_partition_example
)
```

Finally, the build of this Secure Partition should be added to `<TF-M base folder>/secure_fw/partitions/CMakeLists.txt`.

```
add_subdirectory(example)
```

Update the Config System

If the Secure Partition has the build config to enable or disable it, the config option should be added to config systems.

CMake Config

The default value of the config option should be added to the `<TF-M base folder>/config/config_base.cmake`.

```
set(TFM_PARTITION_EXAMPLE  OFF  CACHE BOOL  "Enable the example partition")
```

Kconfig

A `menuconfig` should be added to `<TF-M base folder>/secure_fw/partitions/example/Kconfig`.

```
menuconfig TFM_PARTITION_EXAMPLE
    bool "Enable the Example Partition"
    default n
```

And add it to `<TF-M base folder>/secure_fw/partitions/Kconfig`

```
rsource ``example/Kconfig``
```

Note: The Secure Partition building should be skipped if it is not enabled. This should be done by adding the following code at the begining of its `CMakeLists.txt`

```
if (NOT TFM_PARTITION_EXAMPLE)
    return()
endif()
```

Implement the RoT services

To implement RoT services, the partition needs a source file which contains the implementations of the services, as well as the partition entry point. The user can create this source file under `<TF-M base folder>/secure_fw/partitions/example/tfm_example_partition.c`.

As an example, the RoT service with SID **ROT_A** will be implemented.

Entry point for IPC Model Partitions

This function must have a loop that repeatedly waits for input signals and then processes them, following the Secure Partition initialization.

```
#include "psa_manifest/tfm_example.h"
#include "psa/service.h"

void tfm_example_main(void)
{
    psa_signal_t signals = 0;

    /* Secure Partition initialization */
    example_init();

    /*
     * Continually wait for one or more of the partition's RoT Service or
     * interrupt signals to be asserted and then handle the asserted
     * signal(s).
     */
    while (1) {
        signals = psa_wait(PSA_WAIT_ANY, PSA_BLOCK);
        if (signals & ROT_A_SIGNAL) {
            rot_A();
        } else {
            /* Should not come here */
            psa_panic();
        }
    }
}
```

Entry init for SFN Model Partitions

In the SFN model, the Secure Partition consists of one optional initialization function, which is declared as the `entry_init` symbol as mentioned in section *Add manifest*. After initialization, the entry_init function returns the following values:

- Return `PSA_SUCCESS` if initialization succeeds.
- Return `PSA_SUCCESS` if initialization is partially successful, and you want some SFNs to receive messages. RoT Services that are non-operational must respond to connection requests with `PSA_ERROR_CONNECTION_REFUSED`.
- Return an error status if the initialization failed, and no SFNs within the Secure Partition must be called.

Service implementation for IPC Model

The service is implemented by the `rot_A()` function, which is called upon an incoming signal. This implementation is up to the user, however an example service has been included for reference. The following example sends a message “Hello World” when called.

```c
#include "psa_manifest/tfm_example.h"
#include "psa/service.h"

/* Some other type of services. */
#define SOME_ROT_A_SERVICE_TYPE                (1)

static void rot_A(void)
{
    const int BUFFER_LEN = 32;
    psa_msg_t msg;
    int i;
    uint8_t rec_buf[BUFFER_LEN];
    uint8_t send_buf[BUFFER_LEN] = "Hello World";

    psa_get(ROT_A_SIGNAL, &msg);
    switch (msg.type) {
    case PSA_IPC_CONNECT:
    case PSA_IPC_DISCONNECT:
        /*
         * This service does not require any setup or teardown on connect
         * or disconnect, so just reply with success.
         */
        psa_reply(msg.handle, PSA_SUCCESS);
        break;
    default:
        /* Handling services requested by psa_call. */
        if (msg.type == PSA_IPC_CALL) {
            for (i = 0; i < PSA_MAX_IOVEC; i++) {
                if (msg.in_size[i] != 0) {
                    psa_read(msg.handle, i, rec_buf, BUFFER_LEN);
                }
                if (msg.out_size[i] != 0) {
                    psa_write(msg.handle, i, send_buf, BUFFER_LEN);
                }
            }
            psa_reply(msg.handle, PSA_SUCCESS);
        } else if (msg.type == SOME_ROT_A_SERVICE_TYPE) {
            /* Operations for SOME_ROT_A_SERVICE_TYPE */
        } else {
            /* Invalid type for this Secure Partition. */
            return PSA_ERROR_PROGRAMMER_ERROR;
        }
    }
}
```

Service implementation for SFN Model

SFN model consists of a set of Secure Functions (SFN), one for each RoT Service. The connection, disconnection and request messages do not cause a Secure Partition signal to be asserted for SFN Secure Partitions. Instead, the Secure Function (SFN) for the RoT Service is invoked by the framework, with the message details provided as a parameter to the SFN. To add a secure function (SFN) to process messages for each RoT Service, each SFN will have following prototype.

```
psa_status_t <<name>>_sfn(const psa_msg_t *msg);
```

A connection-based example service has been included for reference which sends a message "Hello World" when called.

```
#include "psa_manifest/tfm_example.h"
#include "psa/service.h"

/* Some other type of services. */
#define SOME_ROT_A_SERVICE_TYPE                (1)

psa_status_t rot_a_sfn(const psa_msg_t *msg)
{
    const int BUFFER_LEN = 32;
    int i;
    uint8_t rec_buf[BUFFER_LEN];
    uint8_t send_buf[BUFFER_LEN] = "Hello World";

    switch (msg->type) {
    case PSA_IPC_CONNECT:
    case PSA_IPC_DISCONNECT:
        /*
         * This service does not require any setup or teardown on connect
         * or disconnect, so just reply with success.
         */
        return PSA_SUCCESS;
    default:
        /* Handling services requested by psa_call. */
        if (msg->type == PSA_IPC_CALL) {
            for (i = 0; i < PSA_MAX_IOVEC; i++) {
                if (msg->in_size[i] != 0) {
                    psa_read(msg->handle, i, rec_buf, BUFFER_LEN);
                }
                if (msg.->out_size[i] != 0) {
                    psa_write(msg->handle, i, send_buf, BUFFER_LEN);
                }
            }
            return PSA_SUCCESS;
        } else if (msg->type == SOME_ROT_A_SERVICE_TYPE) {
            /* Operations for SOME_ROT_A_SERVICE_TYPE */
        } else {
            /* Invalid type for this Secure Partition. */
            return PSA_ERROR_PROGRAMMER_ERROR;
        }
    }
}
```

Test suites and test partitions

A regression test suite can be added to verify whether the new secure partition works as expected. Refer to Adding TF-M Regression Test Suite for the details of adding a regression test suite.

Some regression tests require a dedicated RoT service. The implementations of the RoT service for test are similar to secure partition addition. Refer to Adding partitions for regression tests to get more information.

Out-of-tree Secure Partition build

TF-M supports out-of-tree Secure Partition build, whose source code folders are maintained outside TF-M repo. Developers can configure `TFM_EXTRA_MANIFEST_LIST_FILES` and `TFM_EXTRA_PARTITION_PATHS` in build command line to include out-of-tree Secure Partitions.

- `TFM_EXTRA_MANIFEST_LIST_FILES`

  A list of the absolute path(s) of the manifest list(s) provided by out-of-tree Secure Partition(s). Use semicolons `;` to separate multiple manifest lists. Wrap the multiple manifest lists with double quotes.

- `TFM_EXTRA_PARTITION_PATHS`

  A list of the absolute directories of the out-of-tree Secure Partition source code folder(s). TF-M build system searches `CMakeLists.txt` of partitions in the source code folder(s). Use semicolons `;` to separate multiple out-of-tree Secure Partition directories. Wrap the multiple directories with double quotes.

A single out-of-tree Secure Partition folder can be organized as the figure below.

```
secure partition folder
      ├── CMakeLists.txt
      ├── manifest_list.yaml
      ├── out_of_tree_partition_manifest.yaml
      └── source code
```

In the example above, `TFM_EXTRA_MANIFEST_LIST_FILES` and `TFM_EXTRA_PARTITION_PATHS` in the build command can be configured as listed below.

```
-DTFM_EXTRA_MANIFEST_LIST_FILES=<Absolute-path-sp-folder/manifest_list.yaml>
-DTFM_EXTRA_PARTITION_PATHS=<Absolute-path-sp-folder>
```

Multiple out-of-tree Secure Partitions can be organized in diverse structures. For example, multiple Secure Partitions can be maintained under the same directory as shown below.

```
top-level folder
      ├── Partition 1
      │       ├── CMakeLists.txt
      │       ├── partition_1_manifest.yaml
      │       └── source code
      ├── Partition 2
      │       └── ...
      ├── Partition 3
      │       └── ...
      ├── manifest_list.yaml
      └── Root CMakeLists.txt
```

In the example above, a root CMakeLists.txt includes all the partitions' CMakLists.txt, for example via `add_subdirectory()`. The manifest_list.yaml lists all partitions' manifest files.

TFM_EXTRA_MANIFEST_LIST_FILES and TFM_EXTRA_PARTITION_PATHS in build command line can be configured as listed below.

```
-DTFM_EXTRA_MANIFEST_LIST_FILES=<Absolute-path-top-level-folder/manifest_list.yaml>
-DTFM_EXTRA_PARTITION_PATHS=<Absolute-path-top-level-folder>
```

Alternatively, out-of-tree Secure Partitions can be separated in different folders.

```
partition 1 folder                    partition 2 folder
     ├── CMakeLists.txt                    ├── CMakeLists.txt
     ├── manifest_list.yaml                ├── manifest_list.yaml
     ├── partition_1_manifest.yaml         ├── partition_2_manifest.yaml
     └── source code                       └── source code
```

In the example above, each Secure Partition manages its own manifest files and CMakeLists.txt. TFM_EXTRA_MANIFEST_LIST_FILES and TFM_EXTRA_PARTITION_PATHS in build command line can be configured as listed below. Please note those input shall be wrapped with double quotes.

```
-DTFM_EXTRA_MANIFEST_LIST_FILES="<Absolute-path-part-1-folder/manifest_list.yaml>;
↪<Absolute-path-part-2-folder/manifest_list.yaml>"
-DTFM_EXTRA_PARTITION_PATHS="<Absolute-path-part-1-folder>;<Absolute-path-part-2-folder>"
```

Note: Manifest list paths in TFM_EXTRA_MANIFEST_LIST_FILES do NOT have to be one-to-one mapping to Secure Partition directories in TFM_EXTRA_PARTITION_PATHS. The orders don't matter either.

TFM_EXTRA_MANIFEST_LIST_FILES and TFM_EXTRA_PARTITION_PATHS can be configurated in multiple extra sources. It is recommended to use CMake list APPEND method to avoid unexpected override.

Further Notes

- In the IPC model, Use PSA FF proposed memory accessing mechanism. SPM provides APIs and checking between isolation boundaries, a free accessing of memory can cause program panic.
- In the IPC model, the memory checking inside partition runtime is unnecessary. SPM handles the checking while memory accessing APIs are called.
- In the IPC model, the client ID had been included in the message structure and secure partition can get it when calling psa_get() function. The secure partition does not need to call tfm_core_get_caller_client_id() to get the caller client ID anymore.
- In the IPC model, SPM will check the security policy and partition dependence between client and service. So the service does not need to validate the secure caller anymore.

Reference

PSA Firmware Framework specification
Firmware Framework for M 1.1 Extensions

10.9.7 TF-M Manifest Tool User Guide

This document describes the TF-M manifest tool and its usage. The target audiences are mainly platform integrators and Secure Partition developers.

This document assumes that audiences have knowledge about the terminologies defined in PSA Firmware Framework (FF-M) v1.0[1] and the FF-M v1.1 extensions[2], such as Secure Partition and manifests.

Introduction

The TF-M manifest tool is a python script which is used to parse Secure Partition manifests and generate source header files defined by FF-M[Page 264, 1] such as `psa_manifest/pid.h` and `psa_manifest/sid.h`. It also generates TF-M specific files for building.

In the TF-M build system, the manifest tool is invoked during building automatically. Arguments are passed through CMake variables and they are customizable. See *Usage in TF-M Build System* for more details.

It can be also used as a standalone tool.

Arguments

The tool takes 5 arguments at most.

```
tfm_parse_manifest_list.py [-h] -o out_dir
                           -m manifest list [manifest list ...]
                           -f file-list [file-list ...]
                           -c config-files [config-files ...]
                           [-q]
```

-o/–outdir

Required.

The directory to hold the output files.

-m/–manifest-lists

Required. At least one item must be provided.

A list of TF-M Secure Partition manifest lists which contain one or more Secure Partition manifests in each. See *Manifest List* for details.

-f/–file-list

Required. At least one item must be provided.

A list of files that describe what files the tool should generate. See *Generated File List* for details.

-c/–config-files

Required. At least one item must be provided.

A list of header files which contain build configurations such as isolation level and Secure Partition enabled status. See *Configuration Header File* for details.

-q/–quiet

Optional.

[1] FF-M v1.0 Specification

[2] FF-M v1.1 Extension

Reduces log messages, only warnings and errors will be printed.

Manifest List

A manifest list is a YAML[3] file that describes a list of Secure Partition manifests. Refer to the TF-M default manifest list[4] as an example.

Each manifest list must contain a `manifest_list` attribute which collects the descriptions of Secure Partition manifests. Following are the supported attributes of in the manifest lists.

- `description`

  Required.

  Descriptive information for the tool to refer to the Secure Partition, for example in logs.

- `manifest`

  Required.

  The manifest file of the Secure Partition. Both absolute path and relative path to the manifest list are supported. Environment variables are supported.

- `output_path`

  Optional.

  The directory to hold the Secure Partition specific output files. See *Generated File List* for more details.

  Both absolute path and relative path to the directory specified by `-o/--outdir` are supported. Environment variables are supported.

  It is set to the directory specified by `-o/--outdir` if omitted.

- `conditional`

  Optional.

  The configuration to enable or disable this Secure Partition. The value must be defined in one of the *Configuration Header File*. If it is omitted, the Secure Partition is always enabled.

- `pid`

  Optional.

  The Secure Partition ID (PID).

  For non-test purpose Secure Partitions, it is required. See *Adding Secure Partition* for the PID allocations.

  For test purpose Secure Partitions, this attribute is optional. The manifest tool assigns one for the Secure Partition. The value is not guaranteed to the same for different builds.

- `linker_pattern`

  Optional.

  The information for linker to place the symbols of the Secure Partition. It is only required if you are using the linker scripts provided by TF-M. Each Secure Partition is expected to be built as a library. The name of the library must follow the format of `tfm_<type>_partition_<name>`. The valid value for `<type>` is `[psa_rot, app_rot]` corresponding to the type of the Secure Partitions. The `<name>` is any string to distinguish the Secure Partition from others.

  Supported patterns are:

[3] YAML

[4] TF-M manifest list

  - library_list, must be *tfm_*partition_<name>.*.
  - object_list

    Any object files containing symbols belonging to the Secure Partition that are not included in the Secure Partitions library.

- non_ffm_attributes

  Optional.

  TF-M defines some special manifest attributes for TF-M dedicated Secure Partitions. Those special attributes are not compliant to FF-M. Using the specific attributes requires explicit registration by adding them to this non_ffm_attributes. The purpose is to ensure that developers are aware of the compliance issue. The manifest tool will report errors if unregistered Non-FFM attributes are detected. This attribute is for TF-M specific Secure Partitions and using TF-M-specific attributes is not encouraged.

Generated File List

A generated file list is a YAML file that describes the files to be generated by the manifest tool. Refer to TF-M default generated file list[5] as an example.

Each one must contain a file_list attribute which collects the files to generate. Each item in the file_list must contain the following attributes.

- template

  This attribute is the file path of a Jinja2[6] template. The TF-M manifest tool uses Jinja2 template engine for file generations. It can be a relative path to TF-M root directory or an absolute path. Environment variables are supported.

- output

  The output file of the template. Both absolute path and relative path to the directory specified by -o/--outdir are supported. Environment variables are supported.

The tfm_generated_file_list.yaml is essential to build TF-M.

There are several files that are required for each Secure Partition, so they are not in any generated file lists since one template generates multiple files.

- psa_manifest/<manifestfilename>.h

  manifestfilename is the file name of the manifest. This file contains internal definitions for the Secure Partition implementation, such as RoT Service signals and Secure Functions. Refer to FF-M[1] for more details. The corresponding template is manifestfilename.template

- intermedia_<manifestfilename>.c

  TF-M specific, which holds the stacks of Secure Partitions. This file must be built with the Secure Partition libraries. The corresponding template is partition_intermedia.template.

- load_info_<manifestfilename>.c

  TF-M specific, which contains the load information of Secure Partitions. This file must be built with the TF-M SPM library. The corresponding template is partition_load_info.template.

These files are generated to output_path specified by each Secure Partition in the manifest lists.

[5] TF-M generated file list
[6] Jinja2

Configuration Header File

The format of each configuration item must be

```
#define CONFIG_NAME   VALUE
```

The following format is also supported for boolean type configurations.

```
#define CONFIG_NAME
```

The configurations can be divided into two categories.

- Generic configurations:
  - PSA_FRAMEWORK_ISOLATION_LEVEL

    The isolation level, required. Valid values are [1, 2, 3].
  - CONFIG_TFM_SPM_BACKEND

    The backend of SPM, required. Valid values are [SFN, IPC]. See *SPM backends* for details of backends.
- Secure Partition enablement configurations

  Configurations used to enable or disable Secure Partitions. The configuration names must match the values of conditional attributes in *Manifest List*. Valid values are [0, 1]. It's optional for a Secure Partition which does not have the conditional attribute.

The configurations can be split to multiple files corresponding to the multiple manifest lists.

Usage in TF-M Build System

In the TF-M build system, the manifest tool is invoked during building automatically. The arguments can be customized by altering the CMake configurations.

The manifest lists are passed to the manifest tool via the TFM_MANIFEST_LIST CMake configuration. The default value is the tfm_manifest_list.yaml. It can be overridden or appended with other manifest lists.

Corresponding manifest lists of test Secure Partitions are appended if either TF-M regression or PSA compliance tests are enabled.

The generated file lists are passed via GENERATED_FILE_LISTS. It can be also overridden or appended with other file lists.

The -q argument is appended if PARSE_MANIFEST_QUIET_FLAG is enabled.

Paths in manifest lists and generated file lists can have CMake variables as long as the are absolute paths. The lists then must be processed by the CMake command configure_file()[7] before passing to the manifest tool.

The configuration header file is generated by the build system automatically.

[7] CMake configure_file()

References

10.10 How to build TF-M

Follow the *Build instructions*.

10.11 How to export files for building non-secure applications

Explained in the *Build instructions*.

10.12 How to add a new platform

Porting TF-M to a New Hardware contains guidance on how to add a new platform.

10.13 How to integrate another OS

10.13.1 OS migration to Armv8-M platforms

To work with TF-M on Armv8-M platforms, the OS needs to support the Armv8-M architecture and, in particular, it needs to be able to run in the non-secure world. More information about OS migration to the Armv8-M architecture can be found in the *OS requirements*. Depending upon the system configuration this may require configuring drivers to use appropriate address ranges.

10.13.2 Interface with TF-M

The files needed for the interface with TF-M are exported at the `<install_dir>/interface` path. The NS side is only allowed to call TF-M secure functions (veneers) from the NS Thread mode.

TF-M interface header files are exported in `<install_dir>/interface/include` directory. For example, the Protected Storage (PS) service PSA API is declared in the file `<install_dir>/interface/include/psa/protected_storage.h`.

TF-M also exports a reference implementation of PSA APIs for NS clients in the `<install_dir>/interface/src`.

On Armv8-M TrustZone based platforms, NS OS uses `tfm_ns_interface_dispatch()` to integrate with TF-M implementation of PSA APIs. TF-M provides a reference implementation of this function for RTOS and bare metal use cases. RTOS implementation of `tfm_ns_interface_dispatch()` (provided in `interface\src\os_wrapper\tfm_ns_interface_rtos.c`) uses mutex to provide multithread safety. Mutex wrapper functions defined in `interface/include/os_wrapper/mutex.h` are expected to be provided by NS RTOS. When reference RTOS implementation of dispatch function is used NS application should call `tfm_ns_interface_init()` function before first PSA API call. Bare metal implementation `tfm_ns_interface_dispatch()` (provided in `interface\src\os_wrapper\tfm_ns_interface_bare_metal.c`) does not provide multithread safety and does not require implementation of mutex interface. If needed, instead of using reference implementation, NS application may provide its own implementation of `tfm_ns_interface_dispatch()` function.

TF-M provides a reference implementation of NS mailbox on multi-core platforms, under folder `interface/src/multi_core`. See *Mailbox design* for TF-M multi-core mailbox design.

10.13.3 Interface with non-secure world regression tests

A non-secure application that wants to run the non-secure regression tests needs to call the `tfm_non_secure_client_run_tests()`. This function is exported into the header file `test_framework_integ_test.h` inside the `<build_dir>/install` folder structure in the test specific files, i.e. `<build_dir>/install/export/tfm/test/inc`. The non-secure regression tests are precompiled and delivered as a static library which is available in `<build_dir>/install/export/tfm/test/lib`, so that the non-secure application needs to link against the library to be able to invoke the `tfm_non_secure_client_run_tests()` function. The PS non-secure side regression tests rely on some OS functionality e.g. threads, mutexes etc. These functions comply with CMSIS RTOS2 standard and have been exported as thin wrappers defined in `os_wrapper.h` contained in `<build_dir>/install/export/tfm/test/inc`. OS needs to provide the implementation of these wrappers to be able to run the tests.

10.13.4 NS client Identification

The NS client identification (NSID) is specified by either SPM or NSPE RTOS. If SPM manages the NSID (default option), then the same NSID (-1) will be used for all connections from NS clients. For the case that NSPE RTOS manages the NSID and/or different NSIDs should be used for different NS clients. See *Non-secure Client Extension Integration Guide*.

10.14 Non-secure interrupts

Non-secure interrupts are allowed to preempt Secure thread mode. With the current implementation, a NSPE task can spoof the identity of another NSPE task. This is an issue only when NSPE has provisions for task isolation. Note, that `AIRCR.PRIS` is still set to restrict the priority range available to NS interrupts to the lower half of available priorities so that it wouldn't be possible for any non-secure interrupt to preempt a higher-priority secure interrupt.

10.15 Secure interrupts and scheduling

To ensure correct operation in the general case, the secure scheduler is not run after handling a secure interrupt that pre-empted the NSPE. On systems with specific constraints, it may be desirable to run the scheduler in this situation, which can be done by setting `CONFIG_TFM_SCHEDULE_WHEN_NS_INTERRUPTED` to 1. This could be done if the NSPE is known to be a simple, single-threaded application or if non-secure interrupts cannot pre-empt the SPE, for example.

10.16 Integration with non-Cmake systems

10.16.1 Generated Files

Files that are derived from PSA manifests are generated at build-time by cmake. For integration with systems that do no use cmake, the files must be generated manually.

The `tools/tfm_parse_manifest_list.py` script can be invoked manually. Some arguments will be needed to be provided. Please refer to `tfm_parse_manifest_list.py --help` for more details.

Some variables are used in the template files, these will need to be set in the environment before the script will succeed when the script is not run via cmake.

CHAPTER

ELEVEN

DESIGN DOCUMENTS

11.1 Secure boot

11.1.1 BL1 Immutable bootloader

Author
Raef Coles

Organization
Arm Limited

Contact
raef.coles@arm.com

Introduction

Some devices that use TF-M will require initial boot code that is stored in ROM. There are a variety of reasons that this might happen:

- The device cannot access flash memory without a driver, so needs some setup to be done before main images on flash can be booted.
- The device has no on-chip secure flash, and therefore cannot otherwise maintain a tamper-resistant root of trust.
- The device has a security model that requires an immutable root of trust

Henceforth any bootloader stored in ROM will be referred to as BL1, as it would necessarily be the first stage in the boot chain.

TF-M provides a reference second-stage flash bootloader BL2, in order to allow easier integration. This bootloader implements all secure boot functionality needed to provide a secure chain of trust.

A reference ROM bootloader BL1 has now being added with the same motivation - allowing easier integration of TF-M for platforms that do not have their own BL1 and require one.

BL1 Features and Motivations

The reference ROM bootloader provides the following features:

- A split between code being stored in ROM and in other non-volatile memory.
  - This can allow significant cost reduction in fixing bugs compared to ROM-only bootloaders.
- A secure boot mechanism that allows upgrading the next boot stage (which would usually be BL2).
  - This allows for the fixing of any bugs in the BL2 image.
  - Alternately, this could allow the removal of BL2 in some devices that are constrained in flash space but have ROM.
- A post-quantum resistant asymmetric signature scheme for verifying the next boot stage image.
  - This can allow devices to be securely updated even if attacks involving quantum computers become viable. This could extend the lifespans of devices that might be deployed in the field for many years.
- A mechanism for passing boot measurements to the TF-M runtime so that they can be attested.
- Tooling to create and sign images.
- Fault Injection (FI) and Differential Power Analysis (DPA) mitigations.

BL1_1 and BL1_2 split bootloaders

BL1 is split into two distinct boot stages, BL1_1 which is stored in ROM and BL1_2 which is stored in other non-volatile storage. This would usually be either trusted or untrusted flash, but on platforms without flash memory can be OTP. As BL1_2 is verified against a hash stored in OTP, it is immutable after provisioning even if stored in mutable storage.

Bugs in ROM bootloaders usually cannot be fixed once a device is provisioned / in the field, as ROM code is immutable the only option is fixing the bug in newly manufactured devices.

However, it can be very expensive to change the ROM code of devices once manufacturing has begun, as it requires changes to the photolithography masks that are used to create the device. This cost varies depending on the complexity of the device and of the process node that it is being fabricated on, but can be large, both in engineering time and material/process costs.

By placing the majority of the immutable bootloader in other storage, we can mitigate the costs associated with changing ROM code, as a new BL1_2 image can be used at provisioning time with minimal changeover cost. BL1_1 contains a minimal codebase responsible mainly for the verification of the BL1_2 image.

The bootflow is as follows. For simplicity this assumes that the boot stage after BL1 is BL2, though this is not necessarily the case:

1) BL1_1 begins executing in place from ROM
2) BL1_1 copies BL1_2 into RAM
3) BL1_1 verifies BL1_2 against the hash stored in OTP
4) BL1_1 jumps to BL1_2, if the hash verification has succeeded
5) BL1_2 copies the primary BL2 image from flash into RAM
6) BL1_2 verifies the BL2 image using asymmetric cryptography
7) If verification fails, BL1_2 repeats 5 and 6 with the secondary BL2 image
8) BL1_2 jumps to BL2, if either image has successfully verified

Note: The BL1_2 image is not encrypted, so if it is placed in untrusted flash it will be possible to read the data in the image.

Some optimizations have been made specifically for the case where BL1_2 has been stored in OTP:

OTP can be very expensive in terms of chip area, though new technologies like antifuse OTP decrease this cost. Because of this, the code size of BL1_2 has been minimized. Code-sharing has been configured so that BL1_2 can call functions stored in ROM. Care should be taken that OTP is sized such that it is possible to include versions of the functions used via code-sharing, in case the ROM functions contain bugs, though less space is needed than if all code is duplicated as it is assumed that most functions will not contain bugs.

As OTP memory frequently has low performance, BL1_2 is copied into RAM before it it is executed. It also copies the next image stage into RAM before authenticating it, which allows the next stage to be stored in untrusted flash. This requires that the device have sufficient RAM to contain both the BL1_2 image and the next stage image at the same time. Note that this is done even if BL1_2 is located in XIP-capable flash, as it both allows the use of untrusted flash and simplifies the image upgrade logic.

Note: BL1_2 enables TF-M to be used on devices that contain no secure flash, though the ITS service will not be available. Other services that depend on ITS will not be available without modification.

Secure boot / Image upgrade mechanism

BL1_2 verifies the authenticity of the next stage image via asymmetric cryptography, using a public key that is provisioned into OTP.

BL1_2 implements a rollback protection counter in OTP, which is used to prevent the next stage image being downgraded to a less secure version.

BL1_2 has two image slots, which allows image upgrades to be performed. The primary slot is always booted first, and then if verification of this fails (either due to an invalid signature or due to a version lower than the rollback protection counter) the secondary slot is then booted (subject to the same checks).

BL1_2 contains no image upgrade logic, in order for OTA of the next stage image to be implemented, a later stage in the system must handle downloading new images and placing them in the required slot.

Post-Quantum signature verification in BL1_2

BL1_2 uses a post-quantum asymmetric signature scheme to verify the next stage. The scheme used is Leighton-Michaeli Signatures (henceforth LMS). LMS is standardised in NIST SP800-208 and IETF RFC8554.

LMS is a stateful-hash signature scheme, meaning that:

1) It is constructed from a cryptographic hash function, in this case SHA256.

   - This function can be accelerated by existing hardware accelerators, which can make LMS verification relatively fast compared to other post-quantum signature schemes that cannot be accelerated in hardware yet.

2) Each private key can only be used to sign a certain number of images.

   - BL1_2 uses the SHA256_H10 parameter set, meaning each key can sign 1024 images.

The main downside, the limited amount of possible signatures, can be mitigated by limiting the amount of image upgrades that are done. As BL2 is often currently not upgradable, it is not anticipated that this limit will be problematic. If BL1 is being used to directly boot a TF-M/NS combined image, the limit is more likely to be problematic, and care should be taken to examine the likely update amount.

LMS public keys are 32 bytes in size, and LMS signatures are 1912 bytes in size. The signature size is larger than some asymmetric schemes, though most devices should have enough space in flash to accommodate this.

The main upside of LMS, aside from the security against attacks involving quantum computers, is that it is relatively simple to implement. The software implementation that is used by BL1 is ~3KiB in size, which is considerably smaller than the corresponding RSA implementation which is at least 6.5K. This simplicity of implementation is useful to avoid bugs.

BL1 will use MbedTLS as the source for its implementation of LMS.

Note: As of the time of writing, the LMS code is still in the process of being merged into MbedTLS, so BL1 currently does not support asymmetric verification of the next boot stage. Currently, the next boot stage is hash-locked, so cannot be upgraded.

The Github pull request for LMS can be found here

BL1 boot measurements

BL1 outputs boot measurements in the same format as BL2, utilising the same shared memory area. These measurements can then be included in the attestation token, allowing the attestation of the version of the boot stage after BL1.

BL1 tooling

Image signing scripts are provided for BL1_1 and BL1_2. While the script is named `create_bl2_img.py`, it can be used for any next stage image.

- `bl1/bl1_1/scripts/create_bl1_2_img.py`
- `bl1/bl1_2/scripts/create_bl2_img.py`

These sign (and encrypt in the case of `create_bl2_img.py`) a given image file and append the required headers.

BL1 FI and DPA mitigations

BL1 reuses the FI countermeasures used in the TF-M runtime, which are found in `lib/fih/`.

BL1 implements countermeasures against fault injection. The functions with these countermeasures are found in `bl1/bl1_1/shared_lib/util.c`

`bl_fih_memeql` tests if memory regions have the same value

- It inserts random delays to improve resilience to FIH attacks
- It performs loop integrity checks
- It uses FIH constructs

Using BL1 on new platforms

New platforms must define the following macros in their `region_defs.h`:

- `BL1_1_HEAP_SIZE`
- `BL1_1_STACK_SIZE`
- `BL1_2_HEAP_SIZE`
- `BL1_2_STACK_SIZE`
- `BL1_1_CODE_START`
- `BL1_1_CODE_LIMIT`
- `BL1_1_CODE_SIZE`
- `BL1_2_CODE_START`
- `BL1_2_CODE_LIMIT`
- `BL1_2_CODE_SIZE`
- `PROVISIONING_DATA_START`
- `PROVISIONING_DATA_LIMIT`
- `PROVISIONING_DATA_SIZE`

The `PROVISIONING_DATA_*` defines are used to locate where the data to be provisioned into OTP can be found. These are required as the provisioning bundle needs to contain the entire BL1_2 image, usually >= 8KiB in size, which is too large to be placed in the static data area as is done for all other dummy provisioning data. On development platforms with reprogrammable ROM, this is often placed in unused ROM. On production platforms, this should be located in RAM and then filled with provisioning data. The format of the provisioning data that should be located in the `PROVISIONING_DATA_*` region can be found in `bl1/bl1_1/lib/provisioning.c` in the struct `bl1_assembly_and_test_provisioning_data_t`

If the platform is storing BL1_2 in flash, it must set `BL1_2_IMAGE_FLASH_OFFSET` to the flash offset of the start of BL1_2.

The platform must also implement the HAL functions defined in the following headers:

- `bl1/bl1_1/shared_lib/interface/trng.h`
- `bl1/bl1_1/shared_lib/interface/crypto.h`
- `bl1/bl1_1/shared_lib/interface/otp.h`

If the platform integrates a CryptoCell-312, then it can reuse the existing implementation.

BL1 Testing

New tests have been written to test both the HAL implementation, and the integration of those functions for verifying images. These tests are stored in the `tf-m-tests` repository, under the `test/bl1/` directory, and further subdivided into BL1_1 and BL1_2 tests.

11.1.2 Rollback protection in TF-M secure boot

Author
Tamas Ban

Organization
Arm Limited

Contact
Tamas Ban <tamas.ban@arm.com>

Anti-rollback protection

The goal of anti-rollback protection is to prevent downgrading of the device to an older version of its software, which has been deprecated due to security concerns.

Elements of software upgrade

During a software upgrade the following entities are involved:

- Boot loader
- Manifest data: Metadata of the software image: size, version, hash, signature, etc.
- Software image: binary data, elf, etc.

Validation of new image

Boot loader is responsible to authenticate the new image according to the required policies and decide whether the new image is fulfilling all the requirements. Boot loader verifies the image integrity (hash calculation) and the origination (signature validation), and might verify other requirements as well. If the new image is successfully authenticated then the boot loader is in charge to do the necessary steps (load to execute address, etc.) to enable the new image to be executed. During the validation process the image and the manifest data is checked.

Manifest format in MCUBoot

MCUBoot has a custom manifest format, which is composed of two parts:

  - Image header: Prepended to the beginning of the image. It is integrity protected:
  - TLV section: Appended to the end of the image. It is not integrity protected:
  - Hash of public key, hash of image, signature of image

```
struct image_header {
    uint32_t ih_magic;
    uint32_t ih_load_addr;
    uint16_t ih_hdr_size;
    uint16_t _pad1;
    uint32_t ih_img_size;
    uint32_t ih_flags;
    struct image_version ih_ver;
    uint32_t _pad2;
};
```

Security counter

The aim of a security counter is to have an independent (from the image version) counter in the image manifest to ensure anti-rollback protection. During software release the value of this counter must be increased if a security flaw was fixed in the current version. Later when this image is installed on the device then it is not allowed to go back to earlier versions. It is beneficial to handle this counter independently from image main version number:

- It does not need to increase with each software release
- It makes it possible to do software downgrade to some extent: if the security counter has the same value in the older image then it is accepted.
- If the boot loader verifies multiple images then these can be handled independently.

However, as an alternative solution the image version number also could serve as the security counter of the image. Even the version number itself could be checked during the anti-rollback verification or the value of the security counter could be derived from the image main version. It is the responsibility of the software issuer to define which policy to apply.

Implementation of security counter

The value of the security counter is a security critical data. Any change in its value has a security implication. Therefore it must be in the integrity protected part of the image manifest. Because the image header is almost fully utilized (few unused fields) and the change of image header structure could lead to compatibility issues between boot loader and runtime software, it is proposed to extend the integrity protection to some part of the TLV section. One of the unused fields in the image header can be used to store the size of the integrity protected area of the TLV section. This is necessary to know how to calculate properly the image hash and signature. With this extension of the integrity protected area other attributes that require integrity protection can easily be added to the image manifest.

Trusted non-volatile (NV) counters

The Trusted Base System Architecture (TBSA-M) defines non-volatile (NV) counters or version counters as a counter with the following properties:

- It must only be possible to increment a version counter through a Trusted access.
- It must only be possible to increment a version counter. It must not be possible to decrement it.
- When a version counter reaches its maximum value, it must not roll over, and no further changes must be possible.
- A version counter must be non-volatile, and the stored value must survive a power down period up to the lifetime of the device.

Trusted non-volatile counters can be used to store the value of security counters per updatable software image. Ideally all independently updatable software images should have a separate security counter. In current TF-M implementation the BL2 is not updatable and the secure and non-secure images are updated together as a single binary. Therefore, one counter is enough to implement rollback protection. In future the secure and non-secure binaries will be handled independently; at that time the introduction of a second counter will be necessary.

Currently the NV counters can be manipulated through the interface described in `tfm_plat_nv_counters.h`.

NV counters and anti-rollback protection

Trusted non-volatile counters might not be supported by a hardware platform. In this case anti-rollback protection might still be feasible.

The device threat model needs to consider the following aspects:

- What is the trust level of the boot loader towards the active software
- If the boot loader cannot protect the anti-rollback mechanism from the secure image then the following threat is unmitigated: The content of the memory area which holds the active image could be replaced with a valid but an older version of the software with software only attack, i.e.: remote code execution.
- If the boot loader does not trust the loaded image at all then security counter must have a copy in NV counter area.
- Another aspect to consider is where the active image is stored
- Trusted memory: i.e. on-chip flash (eFlash). The threat that a malicious actor can modify the content of trusted memory with HW attack is out of scope for the current implementation. It is assumed that if an active image and related manifest data is stored in trusted memory then the included security counter cannot be compromised.
- Untrusted memory: i.e. external (off-chip) storage, where malicious actor can physically access it so it is possible to modify its content, therefore the value of included security counter is unreliable.

If the active image is stored in untrusted storage and it is feasible to modify its content (i.e. replace the whole image to an older version including corresponding manifest) then the value of security counter must be copied to the NV counter area. During software validation the boot loader must compare the new image's security counters against the security counter stored in non-volatile counters.

If the active image is stored in trusted memory and boot loader trusts in the active software then it is not mandatory to store the security counter to non-volatile counter area. During software validation the boot loader is allowed to compare the new image's security counters against active image's security counter.

The evaluation of trusted and untrusted memory must be done per target platform during threat modelling.

For the most robust implementation the following principles should be applied:

- Always use NV counters for storing security counter value if supported by the hardware.
- Each software stage must not be able to decrease their corresponding NV counter.

Boot flow with anti-rollback protection

During software upgrade as part of the image validation process the new and active image security counters must be compared. The new image can be accepted if its security counter has a greater or equal value than the active image security counter. From where to extract the active image security counter it can be platform dependent. It might read out directly from active image manifest data (only if it is in trusted memory) or the corresponding non-volatile counter is read.

If non-volatile counters are used to save security counters then their value must be updated:

- If the boot loader does not support to revert previous images (just overwrites the previously active image with the new one) in case of faulty update then the non-volatile counter can be updated to be equal with the new image security counter after successful authentication of the new image.
- If revert is supported then non-volatile counter can be updated just after a test run of the new software when its health check is done. Just in case of successful health check can the counter updated to avoid the prevention of the revert. This might require a secondary restart of the device.

Tool support

There is a Python script, `imgtool.py` which is used to prepare the new image for upgrade (add header, sign the image, append TLV section). This script must be modified to get an additional command line argument which serves as security counter. The security counter must be included in the integrity protected part of the TLV section.

11.1.3 HW crypto key integration in TF-M secure boot

Author
Tamas Ban

Organization
Arm Limited

Contact
Tamas Ban <tamas.ban@arm.com>

Abstract

PSA Trusted Boot and Firmware Update specification requires the support of at least one immutable root of trust public key (ROTPK) for firmware verification. This can be stored using a locked on-chip flash memory, a secure-element or on-chip OTP memory. It also beneficial to be able to provision these keys during the factory life-cycle of the device independently from any software components. The current key handling solution in TF-M secure boot does not supports this key provisioning process. MCUBoot requires compile time built-in public key(s) for image verification. This limitation does not fit well with a scenario with multiple vendors where multiple MCU software components might be deployed by different vendors in different points in the life-cycle of the device and they do not want to share the keys in-advance for embedding in the bootloader code. The goal of this document to propose a solution to decouple MCUBoot from public key(s) and enable the independent deployment of ROTPK.

Existing key handling solution

MCUBoot code contains a compile-time built-in key array which can store any number of key(s) for firmware verification: `bl2/ext/mcuboot/keys.c`. These public key(s) must be available when MCUBoot image is built. There is a script `bl2/ext/mcuboot/scipt/imgtool.py` which can generate a new key pair, and extract the public key part in the expected ASN.1 format and encode it as C structure. The script is also capable of signing the image with the private key. In order to identify and validate the corresponding public key during image verification the hash of the public key is appended to the image manifest area (TLV encoded metadata). During image verification the bootloader retrieves the hash of the public key from the manifest area and compare against the on-the-fly calculated hash value of the built-in public keys. An exact match identifies and validates the public key which must be used for image verification.

Current memory layout:

```
|----------------------|
|                      |
|     MCUBoot code     |
|                      |
|   Full public key    |
|                      |
|----------------------|
```

(continues on next page)

(continued from previous page)

```
|                      |
|       Image          |
|                      |
|----------------------|
|  Image Manifest(TLV) |
|                      |
|  Hash of public key  |
|----------------------|
|                      |
|   Rest of memory     |
|                      |
```

Requirements

- Multiple independent vendor scenario must be supported, when more than one ROTPK is provisioned to the device.
- The corresponding public key for image verification must be identifiable and verifiable.
- Key identity must be immutable and anchored to the device itself.
- Key(s) can be provisioned independently from bootloader.

Design proposal

HW key(s) might stored in OTP memory which is an expensive resource, so storing a large key (such as RSA) is inefficient. Therefore, it is assumed that only the hash of the ROTPK will be stored in the HW. If only the hash of the public key is stored in the HW then the whole public key must be transfered to the device, because it must be available during image verification. This transfer can be done in the same way as how the hash of the key is transfered to the device with the current solution. A new TLV type (TLV_KEY) can be introduced to carry the whole public key. The corresponding public key will be appended to the image itself in the manifest area. It has the drawback that the image will be bigger, but it is assumed that the additional cost of the bigger image (extra flash area + power for download) is less than the cost of the OTP area.

The verification flow:

- Look up the TLV_KEY field to get the public key.
- Calculates its hash(SHA256) value.
- Get the hash of ROTPK from the platform layer (stored in HW)
- Compare the two hash values, if they match then the key can be used to validate the image. In case of failure consider the images as unauthenticated.

Proposed memory layout:

```
|----------------------|
|                      |
|     MCUBoot code     |
|                      |
|    NO PUBLIC KEY     |
|                      |
|----------------------|
|                      |
```

(continues on next page)

(continued from previous page)

```
|       Image          |
|                      |
|----------------------|
|  Image Manifest(TLV) |
|                      |
|   Full public key    |
|----------------------|
|                      |
|                      |
|   Rest of memory     |
|                      |
|                      |
|----------------------|
|   Immutable memory   |
|                      |
|  Hash of public key  |
|----------------------|
|                      |
```

Platform support

A new platform API used by the bootloader must be introduced to retrieve the image corresponding hash of ROTPK:

`enum tfm_plat_err_t tfm_plat_get_rotpk_hash(uint8_t image_id, uint8_t *rotpk_hash, uint32_t *rotpk_hash_size);`

The mapping between image identity and public key can be hard-code in the platform layer. This simplifies the validation of the public key, because no look-up would be needed. It is assumed that the memory area of the ROTPK hash is not directly accessible, therefore a buffer is allocated by the caller to store the hash there.

Compile time configurability

The solution must be compile time configurable in order to be able to switch between built-in key(s) and HW key(s) support, and also due to backwards compatibility.

Tooling

`bl2/ext/mcuboot/scipt/imgtool.py` will be modified to include the whole public key to the TLV area (instead of the hash).

Table to compare the current and proposed solution:

```
|---------|-----------------------|-------------------|--------------------|
|         |Key format in manifest |Key in MCUBoot code|     Key in HW      |
|---------|-----------------------|-------------------|--------------------|
|Proposed |    Full public key    |  No key embedded  | Hash of public key |
|---------|-----------------------|-------------------|--------------------|
|Current  |   Hash of public key  |  Full public key  |   No key in HW     |
|---------|-----------------------|-------------------|--------------------|
```

For secure devices it is security critical to enforce firmware authenticity to protect against execution of malicious software. This is implemented by building a trust chain where each step in the execution chain authenticates the next step before execution. The chain of trust is based on a "Root of Trust" which is implemented using asymmetric cryptography. The Root of Trust is a combination of an immutable bootloader and a public key (ROTPK).

Warning: In order to implement a proper chain of trust functionality, it is mandatory that the first stage bootloader and ROTPK is stored in an **immutable** way. To achieve this the bootloader code must be stored and executed from ROM or such part of flash memory which supports write protection. ROTPK can be stored in a one-time-programmable (OTP) memory. If the SoC has a built-in BL1 (immutable) bootloader and the immutability of TF-M secure boot code is not guaranteed then TF-M secure boot code must be authenticated by BL1 bootloader before execution. If immutability of root of trust (first stage bootloader + ROTPK) is not ensured then there is a risk that the secure boot process could be bypassed, which could lead to arbitrary code execution on the device. Current TF-M secure boot code is intended to be a second stage bootloader, therefore it requires authentication before execution. If TF-M secure boot code is used as a first stage bootloader then it must be stored according to the above requirements.

11.1.4 Second stage bootloader in TF-M

By default, the MCUboot project from GitHub is used as the secure bootloader in TF-M. The repository is going to be automatically downloaded by CMake. The version downloaded can be controlled by the `MCUBOOT_VERSION` CMake variable. If you wish to use a locally downloaded copy, the CMake variable `MCUBOOT_PATH` can be set to its location. This document contains information about how MCUboot has been integrated to TF-M. For further information about MCUboot design please refer to the MCUBoot homepage.

Bootloader is started when CPU is released from reset. It runs in secure mode. It authenticates the firmware image by hash (SHA-256) and digital signature (RSA-3072) validation. Public key, that the checks happens against, can be built into the bootloader image or can be provisioned to the SoC during manufacturing. Metadata of the image is delivered together with the image itself in a header and trailer section. In case of successful authentication, bootloader passes execution to the secure image. Execution never returns to bootloader until next reset.

A default RSA key pair is stored in the repository, public key is in `keys.c` and private key is in `root-RSA-3072.pem`.

Danger: DO NOT use the default keys in a production code, they are exclusively for testing!

The private key must be stored in a safe place outside of the repository. `imgtool.py` (found in the `scripts` directory in the MCUBoot repository, or installed through the pip package manager) can be used to generate new key pairs.

The bootloader can handle the secure and non-secure images independently (multiple image boot) or together (single image boot). In case of multiple image boot they are signed independently with different keys and they can be updated separately. In case of single image boot the secure and non-secure image is handled as a single blob, therefore they must be contiguous in the device memory. In this case they are signed together and also they can be updated only together. In order to have the same artefacts at the end of the build regardless of how the images are handled (independently or together) the images are always concatenated. In case of single image boot they are concatenated first and then signed. In case of multiple image boot they are separately signed first and then concatenated. Preparation of payload is done by Python scripts: `bl2/ext/mcuboot/scripts/`. At the end of a successful build the signed TF-M payload can be found in: `<build_dir>/bin/tfm_s_ns_signed.bin`

11.1.5 Integration with TF-M

MCUBoot assumes a predefined memory layout which is described below (applicable for AN521). It is mandatory to define the primary slot and the secondary slot partitions, but their size and location can be changed:

```
- 0x0000_0000 - 0x0007_FFFF:    BL2 bootloader - MCUBoot
- 0x0008_0000 - 0x000F_FFFF:    Primary slot : Single binary blob:
                                Secure + Non-Secure image;
                                Primary memory partition
  - 0x0008_0000 - 0x0008_03FF:  Common image header
  - 0x0008_0400 - 0x0008_xxxx:  Secure image
  - 0x0008_xxxx - 0x0010_03FF:  Padding (with 0xFF)
  - 0x0010_0400 - 0x0010_xxxx:  Non-secure image
  - 0x0010_xxxx - 0x0010_xxxx:  Hash value(SHA256), RSA signature and other
                                metadata of combined image

- 0x0018_0000 - 0x0027_FFFF:    Secondary slot : Secure + Non-Secure image;
                                Secondary memory partition, structured
                                identically to the primary slot
- 0x0028_0000 - 0x0037_FFFF:    Scratch area, only used during image
                                swapping
```

Multiple image boot requires a slightly different layout:

```
- 0x0000_0000 - 0x0007_FFFF:    BL2 bootloader - MCUBoot
- 0x0008_0000 - 0x000F_FFFF:    Primary slot : Secure image
  - 0x0008_0000 - 0x0008_03FF:  Secure image header
  - 0x0008_0400 - 0x000x_xxxx:  Secure image
  - 0x000x_xxxx - 0x000x_xxxx:  Hash value(SHA256), RSA signature and other
                                metadata of secure image

- 0x0010_0000 - 0x0017_FFFF:    Primary slot : Non-secure image
  - 0x0010_0000 - 0x0010_03FF:  Non-secure image header
  - 0x0010_0400 - 0x001x_xxxx:  Non-secure image
  - 0x001x_xxxx - 0x001x_xxxx:  Hash value(SHA256), RSA signature and other
                                metadata of non-secure image

- 0x0018_0000 - 0x001F_FFFF:    Secondary slot : Secure image
- 0x0020_0000 - 0x0027_FFFF:    Secondary slot : Non-secure image

- 0x0028_0000 - 0x002F_FFFF:    Scratch area, only used during image
                                swapping, used for secure and non-secure
                                image as well
```

11.1.6 Firmware upgrade operation

MCUBoot handles only the firmware authenticity check after start-up and the firmware switch part of the firmware update process. Downloading the new version of the firmware is out-of-scope for MCUBoot. MCUBoot supports three different ways to switch to the new firmware and it is assumed that firmware images are executed-in-place (XIP). The default behaviour is the overwrite-based image upgrade. In this case the active firmware is always executed from the primary slot and the secondary slot is a staging area for new images. Before executing the new firmware image, the content of the primary slot must be overwritten with the content of the secondary slot (the new firmware image). The second option is the image swapping strategy when the content of the two memory slots must be physically swapped. This needs the scratch area to be defined in the memory layout. The third option is the direct execute-in-place version, which eliminates the complexity of image swapping and its administration. Active image can be executed from either memory slot, but new firmware must be linked to the address space of the proper (currently inactive) memory slot.

Overwrite operation

Active image is stored in the primary slot, and this image is started always by the bootloader. Therefore images must be linked to the primary slot. If the bootloader finds a valid image in the secondary slot, which is marked for upgrade, then the content of the primary slot will be simply overwritten with the content of the secondary slot, before starting the new image from the primary slot. After the content of the primary slot has been successfully overwritten, the header and trailer of the new image in the secondary slot is erased to prevent the triggering of another unnecessary image upgrade after a restart. The overwrite operation is fail-safe and resistant to power-cut failures. For more details please refer to the MCUBoot documentation.

Swapping operation

This operation can be set with the `MCUBOOT_UPGRADE_STRATEGY` compile time switch (see *Build time configuration*). With swapping image upgrade strategy the active image is also stored in the primary slot and it will always be started by the bootloader. If the bootloader finds a valid image in the secondary slot, which is marked for upgrade, then contents of the primary slot and the secondary slot will be swapped, before starting the new image from the primary slot. Scratch area is used as a temporary storage place during image swapping. Update mark from the secondary slot is removed when the swapping is successful. The boot loader can revert the swapping as a fall-back mechanism to recover the previous working firmware version after a faulty update. The swap operation is fail-safe and resistant to power-cut failures. For more details please refer to the MCUBoot documentation.

Note: After a successful image upgrade, user can mark the image as “OK” at runtime by explicitly calling `psa_fwu_accept`. When this happens, the swap is made “permanent” and MCUBoot will then still choose to run it during the next boot.

TF-M does not set the image_ok flag, because it is user’s duty to determine whether the image is acceptable. Therefore the bootloader will always perform a “revert” (swap the images back) during the next boot.

Direct execute-in-place operation

This operation can be set with the `MCUBOOT_UPGRADE_STRATEGY` compile time switch (see *Build time configuration*). When enabling direct-xip operation then the active image flag is moved between slots during firmware upgrade. If firmware is executed-in-place (XIP), then two firmware images must be generated. One of them is linked to be executed from the primary slot memory region and the other from the secondary slot. The firmware upgrade client, which downloads the new image, must be aware, which slot hosts the active firmware and which acts as a staging area and it is responsible for downloading the proper firmware image. At boot time MCUBoot inspects the version number in the image header and passes execution to the newer firmware version. New image must be marked for upgrade which is automatically done by Python scripts at compile time. Image verification is done the same way in all operational modes. If new image fails during authentication then MCUBoot erases the memory slot and starts the other image, after successful authentication.

To select which slot the image is to be executed from, set `MCUBOOT_EXECUTION_SLOT` to the desired index. It is suggested that you create two build directories when building images using this mode, as intermediate dependencies cannot be reused due to changes in the flash layout.

Note: Only single image boot is supported with direct-xip upgrade mode.

RAM Loading firmware upgrade

Musca-S supports an image upgrade mode that is separate to the other (overwrite, swapping and dirext-xip) modes. This is the `RAM load` mode (please refer to the table below). Like the direct-xip mode, this selects the newest image by reading the image version numbers in the image headers, but instead of executing it in place, the newest image is copied to RAM for execution. The load address, the location in RAM where the image is copied to, is stored in the image header.

Summary of different modes for image upgrade

Different implementations of the image upgrade operation (whether through overwriting, swapping, direct-xip or loading into RAM and executing from there) are supported by the platforms. The table below shows which of these modes are supported by which platforms:

	Without BL2[1]	With BL2[2]			
	XIP	XIP	XIP	XIP	Not XIP
		Overwrite[3]	Swap[4]	direct-xip[5]	RAM load[6]
AN521	Yes	Yes	Yes	Yes	No
AN519	Yes	Yes	Yes	Yes	No
FVP_SSE300_MPS3	No	Yes	Yes	Yes	No
Corstone-310 FVP	Yes	Yes	Yes	Yes	No
LPC55S69	Yes	Yes	No	Yes	No
Musca-B1	Yes	Yes	Yes	Yes	No
Musca-S1	Yes	Yes	Yes	Yes	No
AN524	Yes	No	No	Yes	No
AN547	No	Yes	Yes	Yes	No
AN552	No	Yes	Yes	Yes	No
PSoC64	Yes	No	No	No	No
STM_DISCO_L562QE	No	Yes	No	No	No
STM_NUCLEO_L552ZE_Q	No	Yes	No	No	No
nRF9160 DK	Yes	Yes	No	No	No
nRF5340 DK	Yes	Yes	No	No	No
BL5340 DVK	Yes	Yes	Yes	No	No
RSE	No	No	No	No	Yes

[1] To disable BL2, please set the `BL2` cmake option to `OFF`

[2] BL2 is enabled by default

[3] The image executes in-place (XIP) and is in Overwrite mode for image update by default

[4] To enable XIP Swap mode, assign the "SWAP_USING_SCRATCH" or "SWAP_USING_MOVE" string to the `MCUBOOT_UPGRADE_STRATEGY` configuration variable in the build configuration file, or include this macro definition in the command line

[5] To enable direct-xip, assign the "DIRECT_XIP" string to the `MCUBOOT_UPGRADE_STRATEGY` configuration variable in the build configuration file, or include this macro definition in the command line

[6] To enable RAM load, assign the "RAM_LOAD" string to the `MCUBOOT_UPGRADE_STRATEGY` configuration variable in the build configuration

11.1.7 Multiple image boot

It is possible to update the firmware images independently to support the scenario when secure and non-secure images are provided by different vendors. Multiple image boot is supported only together with the overwrite and swap firmware upgrade modes.

It is possible to describe the dependencies of the images on each other in order to avoid a faulty upgrade when incompatible versions would be installed. These dependencies are part of the image manifest area. The dependencies are composed from two parts:

- **Image identifier:** The number of the image which the current image (whose manifest area contains the dependency entry) depends on. The image identifier starts from 0.
- **Minimum version:** The minimum version of other image must be present on the device by the end of the upgrade (both images might be updated at the same time).

Dependencies can be added to the images at compile time with the following compile time switches:

- `MCUBOOT_S_IMAGE_MIN_VER` It is added to the non-secure image and specifies the minimum required version of the secure image.
- `MCUBOOT_NS_IMAGE_MIN_VER` It is added to the secure image and specifies the minimum required version of the non-secure image.

Example of how to provide the secure image minimum version:

```
cmake -DTFM_PLATFORM=arm/musca_b1 -DMCUBOOT_S_IMAGE_MIN_VER=1.2.3+4 ..
```

11.1.8 Signature algorithms

MbedTLS library is used to sign the images. The list of supported signing algorithms:

- *RSA-2048*
- *RSA-3072*: default

Example keys stored in:

- `root-RSA-2048.pem` : Used to sign single image (S+NS) or secure image in case of multiple image boot
- `root-RSA-2048_1.pem` : Used to sign non-secure image in case of multiple image boot
- `root-RSA-3072.pem` : Used to sign single image (S+NS) or secure image in case of multiple image boot
- `root-RSA-3072_1.pem` : Used to sign non-secure image in case of multiple image boot

file, or include this macro definition in the command line

11.1.9 Build time configuration

MCUBoot related compile time switches can be set by cmake variables.

- **BL2 (default: True):**
    - **True:** TF-M built together with bootloader. MCUBoot is executed after reset and it authenticates TF-M and starts secure code.
    - **False:** TF-M built without bootloader. Secure image linked to the beginning of the device memory and executed after reset. If it is false then using any of the further compile time switches is invalid.
- **MCUBOOT_UPGRADE_STRATEGY (default: "OVERWRITE_ONLY"):**
    - **"OVERWRITE_ONLY":** Default firmware upgrade operation with overwrite.
    - **"SWAP_USING_SCRATCH":** Activate swapping firmware upgrade operation with a scratch area in flash
    - **"SWAP_USING_MOVE":** Activate swapping firmware upgrade operation without a scratch area in flash
    - **"DIRECT_XIP":** Activate direct execute-in-place firmware upgrade operation.
    - **"RAM_LOAD":** Activate RAM loading firmware upgrade operation, where the latest image is copied to RAM and runs from there instead of being executed in-place.
- **MCUBOOT_SIGNATURE_TYPE (default: RSA-3072):**
    - **RSA-2048:** Image is signed with RSA algorithm and signed with 2048 bit key.
    - **RSA-3072:** Image is signed with RSA algorithm and signed with 3072 bit key.
    - **EC-P256:** Image is signed with ECDSA P-256 algorithm.
    - **EC-P384:** Image is signed with ECDSA P-384 algorithm.
- **MCUBOOT_IMAGE_NUMBER (default: 2):**
    - **1:** Single image boot, secure and non-secure images are signed and updated together.
    - **2:** Multiple image boot, secure and non-secure images are signed and updatable independently.
- **MCUBOOT_HW_KEY (default: True):**
    - **True:** The hash of public key is provisioned to the SoC and the image manifest contains the whole public key (imgtool uses `--public_key_format=full`). MCUBoot validates the key before using it for firmware authentication, it calculates the hash of public key from the manifest and compare against the retrieved key-hash from the hardware. This way MCUBoot is independent from the public key(s). Key(s) can be provisioned any time and by different parties.
    - **False:** The whole public key is embedded to the bootloader code and the image manifest contains only the hash of the public key (imgtool uses `--public_key_format=hash`). MCUBoot validates the key before using it for firmware authentication, it calculates the hash of built-in public key and compare against the retrieved key-hash from the image manifest. After this the bootloader can verify that the image was signed with a private key that corresponds to the retrieved key-hash (it can have more public keys embedded in and it may have to look for the matching one). All the public key(s) must be known at MCUBoot build time.
- **MCUBOOT_BUILTIN_KEY (default: False):**
    - **True:** When enabled, the entire public key used for signature verification must be provisioned to the target device. In this case, neither the code nor the image metadata needs to contain any public key data. During image validation only a key ID is passed to the verifier function for the required key to

be selected. The key handling is entirely the responsibility of the underlying crypto library and the details of the key handling mechanism are abstracted away from the boot code.

- **MCUBOOT_LOG_LEVEL:**
  Can be used to configure the level of logging in MCUBoot. The possible values are the following:
  - **OFF**
  - **ERROR**
  - **WARNING**
  - **INFO**
  - **DEBUG**

  The logging in MCUBoot can be disabled and thus the code size can be reduced by setting it to `OFF`. Its value depends on the build type. If the build type is `Debug` then default value is `INFO`. In case of different kinds of `Release` builds the default value is `OFF`. The default value can be overridden through the command line or in the CMake GUI regardless of the build type.
- **MCUBOOT_ENC_IMAGES (default: False):**
  - **True:** Adds encrypted image support in the source and encrypts the resulting image using the `enc-rsa2048-pub.pem` key found in the MCUBoot repository.
  - **False:** Doesn't add encrypted image support and doesn't encrypt the image.

  Note: The decryption takes place during the upgrade process, when the images are being moved between the slots. This means that boards that don't already have an image on them with MCUBoot that has been compiled with `MCUBOOT_ENCRYPT_RSA` enabled need special treatment. In order to load an encrypted image to such boards, an upgrade needs to be executed. This can be done by using MCUBoot, putting an image in the secondary image area, and setting `MCUBOOT_ENCRYPT_RSA` to `ON`. When using the `OVERWRITE_ONLY` upgrade strategy, this is enough. When using `SWAP_USING_SCRATCH` or `SWAP_USING_MOVE`, an image is needed in the primary image area as well, to trigger the update.

  Danger: DO NOT use the `enc-rsa2048-pub.pem` key in production code, it is exclusively for testing!

Image versioning

An image version number is written to its header by one of the Python scripts, and this number is used by the bootloader when the direct execute-in-place or the RAM loading mode is enabled. It is also used in case of multiple image boot when the bootloader checks the image dependencies if any have been added to the images.

The version number of the image (single image boot) can manually be passed in through the command line in the cmake configuration step:

```
cmake -DTFM_PLATFORM=arm/musca_b1 -DIMAGE_VERSION_S=1.2.3+4 ..
```

Alternatively, the version number can be less specific (e.g 1, 1.2, or 1.2.3), where the missing numbers are automatically set to zero. The image version number argument is optional, and if it is left out, then the version numbers of the image(s) being built in the same directory will automatically change. In this case, the last component (the build number) automatically increments from the previous one: 0.0.0+1 -> 0.0.0+2, for as many times as the build is re-ran, **until a number is explicitly provided**. If automatic versioning is in place and then an image version number is provided for the first time, the new number will take precedence and be used instead. All subsequent image versions are then set to the last number that has been specified, and the build number would stop incrementing. Any new version numbers that are provided will overwrite the previous one: 0.0.0+1 -> 0.0.0+2. Note: To re-apply automatic image versioning, please start a clean build without specifying the image version number at all. In case of multiple image boot there are separate compile time switches for both images to provide their version: IMAGE_VERSION_S and IMAGE_VERSION_NS. These must be used instead of IMAGE_VERSION_S.

Security counter

Each signed image contains a security counter in its manifest. It is used by the bootloader and its aim is to have an independent (from the image version) counter to ensure rollback protection by comparing the new image's security counter against the original (currently active) image's security counter during the image upgrade process. It is added to the manifest (to the TLV area that is appended to the end of the image) by one of the Python scripts when signing the image. The value of the security counter is security critical data and it is in the integrity protected part of the image. The last valid security counter should always be stored in a non-volatile and trusted component of the device and its value should always be increased if a security flaw was fixed in the current image version. The value of the security counter (single image boot) can be specified at build time in the cmake configuration step:

```
cmake -DTFM_PLATFORM=arm/musca_b1 -DSECURITY_COUNTER_S=42 ../
```

The security counter can be independent from the image version, but not necessarily. Alternatively, if it is not specified at build time with the SECURITY_COUNTER option the Python script will automatically generate it from the image version number (not including the build number) and this value will be added to the signed image. In case of multiple image boot there are separate compile time switches for both images to provide their security counter value: SECURITY_COUNTER_S and SECURITY_COUNTER_NS. These must be used instead of SECURITY_COUNTER_S. If these are not defined then the security counter values will be derived from the corresponding image version similar to the single image boot.

11.1.10 Signing the images manually

Normally the build system handles the signing (computing hash over the image and security critical manifest data and then signing the hash) of the firmware images. However, the images also can be signed manually by using the imgtool Python program which is located in the MCUboot repository in the scripts folder or can be installed with the pip package manager. Issue the python3 imgtool.py sign --help command in the directory for more information about the mandatory and optional arguments. The tool takes an image in binary or Intel Hex format and adds a header and trailer that MCUBoot is expecting. In case of single image boot after a successful build the tfm_s_ns.bin build artifact (contains the concatenated secure and non-secure images) must be passed to the script and in case of multiple image boot the tfm_s.bin and tfm_ns.bin binaries can be passed to prepare the signed images.

Signing the secure image manually in case of multiple image boot

```
python3 bl2/ext/mcuboot/scripts/imgtool.py sign \
    --layout <build_dir>/bl2/ext/mcuboot/CMakeFiles/signing_layout_s.dir/signing_layout_
↪s.c.obj \
    -k <tfm_dir>/bl2/ext/mcuboot/root-RSA-3072.pem \
    --public-key-format full \
    --align 1 \
    -v 1.2.3+4 \
    -d "(1,1.2.3+0)" \
    -s 42 \
    -H 0x400 \
    <build_dir>/bin/tfm_s.bin \
    <build_dir>/bin/tfm_s_signed.bin
```

11.1.11 Testing firmware upgrade

As downloading the new firmware image is out of scope for MCUBoot, the update process is started from a state where the original and the new image are already programmed to the appropriate memory slots. To generate the original and a new firmware package, TF-M is built twice with different build configurations.

Overwriting firmware upgrade

Run TF-M build twice with `MCUBOOT_IMAGE_NUMBER` set to "1" in both cases (single image boot), but with two different build configurations: default and regression. Save the artifacts between builds, because second run can overwrite original binaries. Download default build to the primary slot and regression build to the secondary slot.

Executing firmware upgrade on FVP_MPS2_AEMv8M

```
<ARM_DS_PATH>/sw/models/bin/FVP_MPS2_AEMv8M  \
--parameter fvp_mps2.platform_type=2 \
--parameter cpu0.baseline=0 \
--parameter cpu0.INITVTOR_S=0x10000000 \
--parameter cpu0.semihosting-enable=0 \
--parameter fvp_mps2.DISABLE_GATING=0 \
--parameter fvp_mps2.telnetterminal0.start_telnet=1 \
--parameter fvp_mps2.telnetterminal1.start_telnet=0 \
--parameter fvp_mps2.telnetterminal2.start_telnet=0 \
--parameter fvp_mps2.telnetterminal0.quiet=0 \
--parameter fvp_mps2.telnetterminal1.quiet=1 \
--parameter fvp_mps2.telnetterminal2.quiet=1 \
--application cpu0=<build_dir>/bin/bl2.axf \
--data cpu0=<default_build_dir>/bin/tfm_s_ns_signed.bin@0x10080000 \
--data cpu0=<regresssion_build_dir>/bin/tfm_s_ns_signed.bin@0x10180000
```

Executing firmware upgrade on SSE 200 FPGA on MPS2 board

```
TITLE: Versatile Express Images Configuration File
[IMAGES]
TOTALIMAGES: 3                     ;Number of Images (Max: 32)
IMAGE0ADDRESS: 0x00000000
IMAGE0FILE: \Software\bl2.axf      ; BL2 bootloader
IMAGE1ADDRESS: 0x10080000
IMAGE1FILE: \Software\tfm_sig1.bin ; TF-M default test binary blob
IMAGE2ADDRESS: 0x10180000
IMAGE2FILE: \Software\tfm_sig2.bin ; TF-M regression test binary blob
```

The following message will be shown in case of successful firmware upgrade:

```
[INF] Starting bootloader
[INF] Swap type: test
[INF] Image upgrade secondary slot -> primary slot
[INF] Erasing the primary slot
[INF] Copying the secondary slot to the primary slot: 0x100000 bytes
[INF] Bootloader chainload address offset: 0x80000
```

(continues on next page)

(continued from previous page)

```
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!

#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_PS_TEST_2XXX)...
...
```

To update the secure and non-secure images separately (multiple image boot), set the `MCUBOOT_IMAGE_NUMBER` switch to “2” (this is the default configuration value) and follow the same instructions as in case of single image boot.

Executing multiple firmware upgrades on SSE 200 FPGA on MPS2 board

```
TITLE: Versatile Express Images Configuration File
[IMAGES]
TOTALIMAGES: 4                     ;Number of Images (Max: 32)
IMAGE0ADDRESS: 0x00000000
IMAGE0FILE: \Software\bl2.axf      ; BL2 bootloader
IMAGE1ADDRESS: 0x10080000
IMAGE1FILE: \Software\tfm_sign.bin ; TF-M default test binary blob
IMAGE2ADDRESS: 0x10180000
IMAGE2FILE: \Software\tfm_ss1.bin  ; TF-M regression test secure (signed) image
IMAGE3ADDRESS: 0x10200000
IMAGE3FILE: \Software\tfm_nss1.bin ; TF-M regression test non-secure (signed) image
```

Note that both the concatenated binary blob (the images are signed separately and then concatenated) and the separate signed images can be downloaded to the device because on this platform (AN521) both the primary slots and the secondary slots are contiguous areas in the Flash (see *Integration with TF-M*). The following message will be shown in case of successful firmware upgrades:

```
[INF] Starting bootloader
[INF] Swap type: test
[INF] Swap type: test
[INF] Image upgrade secondary slot -> primary slot
[INF] Erasing the primary slot
[INF] Copying the secondary slot to the primary slot: 0x80000 bytes
[INF] Image upgrade secondary slot -> primary slot
[INF] Erasing the primary slot
[INF] Copying the secondary slot to the primary slot: 0x80000 bytes
[INF] Bootloader chainload address offset: 0x80000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!
TFM level is: 1
[Sec Thread] Jumping to non-secure code...

#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_PS_TEST_2XXX)...
...
```

Swapping firmware upgrade

Follow the same instructions and platform related configurations as in case of overwriting build including these changes:

- Set the `MCUBOOT_UPGRADE_STRATEGY` compile time switch to "SWAP" before build.
- Set the `MCUBOOT_IMAGE_NUMBER` compile time switch to "1" (single image boot) or "2" (multiple image boot) before build.

During single image boot the following message will be shown in case of successful firmware upgrade, `Swap type: test` indicates that images were swapped:

```
[INF] Starting bootloader
[INF] Image 0: magic= good, copy_done=0x3, image_ok=0x3
[INF] Scratch: magic=  bad, copy_done=0x0, image_ok=0x2
[INF] Boot source: primary slot
[INF] Swap type: test
[INF] Bootloader chainload address offset: 0x80000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!

#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_PS_TEST_2XXX)...
...
```

Direct execute-in-place firmware upgrade

Follow the same instructions and platform related configurations as in case of overwriting build including these changes:

- Set the `MCUBOOT_UPGRADE_STRATEGY` compile time switch to "DIRECT_XIP" before build.
- set `MCUBOOT_EXECUTION_SLOT` to 1 in the regression build dir.
- Make sure the image version number was increased between the two build runs either by specifying it manually or by checking in the build log that it was incremented automatically.

Executing firmware upgrade on FVP_MPS2_AEMv8M

```
<ARM_DS_PATH>/sw/models/bin/FVP_MPS2_AEMv8M  \
--parameter fvp_mps2.platform_type=2 \
--parameter cpu0.baseline=0 \
--parameter cpu0.INITVTOR_S=0x10000000 \
--parameter cpu0.semihosting-enable=0 \
--parameter fvp_mps2.DISABLE_GATING=0 \
--parameter fvp_mps2.telnetterminal0.start_telnet=1 \
--parameter fvp_mps2.telnetterminal1.start_telnet=0 \
--parameter fvp_mps2.telnetterminal2.start_telnet=0 \
--parameter fvp_mps2.telnetterminal0.quiet=0 \
--parameter fvp_mps2.telnetterminal1.quiet=1 \
--parameter fvp_mps2.telnetterminal2.quiet=1 \
--application cpu0=<build_dir>/bin/bl2.axf \
--data cpu0=<default_build_dir>/bin/tfm_s_ns_signed.bin@0x10080000 \
--data cpu0=<regresssion_build_dir>/bin/tfm_s_ns_signed.bin@0x10180000
```

Executing firmware upgrade on SSE 200 FPGA on MPS2 board

```
TITLE: Versatile Express Images Configuration File
[IMAGES]
TOTALIMAGES: 3                     ;Number of Images (Max: 32)
IMAGE0ADDRESS: 0x00000000
IMAGE0FILE: \Software\bl2.axf      ; BL2 bootloader
IMAGE1ADDRESS: 0x10080000
IMAGE1FILE: \Software\tfm_sign.bin ; TF-M default test binary blob
IMAGE2ADDRESS: 0x10180000
IMAGE2FILE: \Software\tfm_sig1.bin ; TF-M regression test binary blob
```

Executing firmware upgrade on Musca-B1 and Musca-S1 boards

After the two images have been built, they can be concatenated to create the combined image using `srec_cat`:

- Linux:

```
srec_cat bin/bl2.bin -Binary -offset 0xA000000 tfm_sign.bin -Binary -offset 0xA020000 tfm_sign_1.bin -Binary -offset 0xA100000 -o tfm.hex -Intel
```

- Windows:

```
srec_cat.exe bin\bl2.bin -Binary -offset 0xA000000 tfm_sign.bin -Binary -offset 0xA020000 tfm_sign_1.bin -Binary -offset 0xA100000 -o tfm.hex -Intel
```

The following message will be shown in case of successful firmware upgrade, notice that image with higher version number (`version=1.2.3.5`) is executed:

```
[INF] Starting bootloader
[INF] Image 0: version=1.2.3.4, magic= good, image_ok=0x3
[INF] Image 1: version=1.2.3.5, magic= good, image_ok=0x3
[INF] Booting image from the secondary slot
[INF] Bootloader chainload address offset: 0xa0000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!

#### Execute test suites for the Secure area ####
Running Test Suite PSA protected storage S interface tests (TFM_PS_TEST_2XXX)...
...
```

Executing firmware upgrade on CoreLink SSE-200 Subsystem for MPS3 (AN524)

```
TITLE: Arm MPS3 FPGA prototyping board Images Configuration File

[IMAGES]
TOTALIMAGES: 3                     ;Number of Images (Max: 32)

IMAGE0UPDATE: AUTO                 ;Image Update:NONE/AUTO/FORCE
IMAGE0ADDRESS: 0x00000000
```

(continues on next page)

(continued from previous page)

```
IMAGE0FILE: \SOFTWARE\bl2.bin      ;BL2 bootloader
IMAGE1UPDATE: AUTO
IMAGE1ADDRESS: 0x00040000
IMAGE1FILE: \SOFTWARE\tfm_sig0.bin ;TF-M example application binary blob
IMAGE2UPDATE: AUTO
IMAGE2ADDRESS: 0x000C0000
IMAGE2FILE: \SOFTWARE\tfm_sig1.bin ;TF-M regression test binary blob
```

RAM loading firmware upgrade

To enable RAM loading, please set `MCUBOOT_UPGRADE_STRATEGY` to “RAM_LOAD” (either in the configuration file or through the command line), and then specify a destination load address in RAM where the image can be copied to and executed from. The `S_IMAGE_LOAD_ADDRESS` macro must be specified in the target dependent files, and if multiple image boot is enabled then `NS_IMAGE_LOAD_ADDRESS` must also be defined. For example with Musca-S, its `flash_layout.h` file in the `platform` folder should include `#define S_IMAGE_LOAD_ADDRESS #0xA0020000`

Executing firmware upgrade on Musca-S board

After two images have been built, they can be concatenated to create the combined image using `srec_cat`:

- Linux:

```
srec_cat bin/bl2.bin -Binary -offset 0xA000000 tfm_sign_old.bin -Binary -offset␣
↪0xA020000 tfm_sign_new.bin -Binary -offset 0xA100000 -o tfm.hex -Intel
```

- Windows:

```
srec_cat.exe bin\bl2.bin -Binary -offset 0xA000000 tfm_sign_old.bin -Binary -offset␣
↪0xA020000 tfm_sign_new.bin -Binary -offset 0xA100000 -o tfm.hex -Intel
```

The following message will be shown in case of successful firmware upgrade when, RAM loading is enabled, notice that image with higher version number (`version=0.0.0.2`) is executed:

```
[INF] Starting bootloader
[INF] Image 0: version=0.0.0.1, magic= good, image_ok=0x3
[INF] Image 1: version=0.0.0.2, magic= good, image_ok=0x3
[INF] Image has been copied from the secondary slot in flash to SRAM address 0xA0020000
[INF] Booting image from SRAM at address 0xA0020000
[INF] Bootloader chainload address offset: 0x20000
[INF] Jumping to the first image slot
[Sec Thread] Secure image initializing!
```

11.1.12 Integration with Firmware Update service

The shim layer of the Firmware Update partition calls the APIs in bootutil_misc.c to control the image status.

- Call `boot_set_pending_multi()` to make the image as a candidate image for booting.
- Call `boot_set_confirmed_multi()` to make the image as a permanent image.

Note: Currently, in direct-xip mode and ram-load mode, TF-M cannot get the information of which slot contains the running image from the bootloader. So the Firmware Update partition cannot decide where to write the new image. As a result, the firmware update service is not supported in direct-xip mode and ram-load mode.

11.2 Dual-CPU

11.2.1 Booting a Dual-Core System

Author
Chris Brand

Organization
Cypress Semiconductor Corporation

Contact
chris.brand@cypress.com

System Architecture

There are many possibly ways to design a dual core system. Some important considerations from a boot perspective are:

- Which core has access to which areas of Flash?
  - It is possible that the secure core has no access to the Flash from which the non-secure core will boot, in which case the non-secure core will presumably have a separate root of trust and perform its own integrity checks on boot.
- How does the non-secure core behave on power-up? Is it held in reset, does it jump to a set address, …?
- What are the performance characteristics of the two core?
  - There could be a great disparity in performance

TF-M Twin Core Booting

In an effort to make the problem manageable, as well as to provide a system with good performance, that is flexible enough to work for a variety of dual core systems, the following design decisions have been made:

- TF-M will (for now) only support systems where the secure core has full access to the Flash that the non-secure core will boot from
  - This keeps the boot flow as close as possible to the single core design, with the secure core responsible for maintaining the chain of trust for the entire system, and for upgrade of the entire system
- The secure code will make a platform-specific call immediately after setting up hardware protection to (potentially) start the non-secure core running
  - This is the earliest point at which it is safe to allow the non-secure code to start running, so starting it here ensures system integrity while also giving the non-secure code the maximum amount of time to perform its initialization
  - Note that this is after the bootloader has validated the non-secure image, which is the other key part to maintain security
  - This also means that only tfm_s and tfm_ns have to change, and not mcuboot
- Both the secure and non-secure code will make platform-specific calls to establish a synchronization point. This will be after both sides have done any initialization that is required, including setting up inter-core communications. On a single core system, this would be the point at which the secure code jumps to the non-secure code, and at the very start of the non-secure code.
- After completing initialization on the secure core (at the point where on a single core system, it would jump to the non-secure code), the main thread on the secure core will be allowed to die
  - The scheduler has been started at this point, and an idle thread exists. Any additional work that is only required in the dual core case will be interrupt-driven.
  - All work related to the non-secure core will take place from a `ns_agent_mailbox` partition, which will establish communication with the non-secure core and then act on its behalf
- Because both cores may be booting in parallel, executing different initialization code, at different speeds, the design must be resilient if either core attempts to communicate with the other before the latter is ready. For example, the client (non-secure) side of the IPC mechanism must be able to handle the situation where it has to wait for the server (secure) side to finish setting up the IPC mechanism.
  - This relates to the synchronization calls mentioned above. It means that those calls cannot utilise the IPC mechanism, but must instead use some platform-specific mechanism to establish this synchronization. This could be as simple as setting aside a small area of shared memory and having both sides set a "ready" flag, but may well also involve the use of interrupts.
  - This also means that the synchronization call must take place after the IPC mechanism has been set up but before any attempt (by either side) to use it.

API Additions

Three new HAL functions are required:

```
void tfm_hal_boot_ns_cpu(uintptr_t start_addr);
```

- Called on the secure core from `ns_agent_mailbox` partition when it first runs (after protections have been configured).
- Performs the necessary actions to start the non-secure core running the code at the specified address.

```
void tfm_hal_wait_for_ns_cpu_ready(void);
```

- Called on the secure core from `ns_agent_mailbox` partition after making the above call.
- Flags that the secure core has completed its initialization, including setting up the IPC mechanism.
- Waits, if necessary, for the non-secure core to flag that it has completed its initialisation

```
void tfm_ns_wait_for_s_cpu_ready(void);
```

- Called on the non-secure core from `main()` after the dual-core-specific initialization (on a single core system, this would be the start of the non-secure code), before the first use of the IPC mechanism.
- Flags that the non-secure side has completed its initialization.
- Waits, if necessary, for the secure core to flag that it has completed its initialization.

For all three, an empty implementation will be provided with a weak symbol so that platforms only have to provide the new functions if they are required.

11.2.2 Communication Prototype Between NSPE And SPE In Dual Core System

Author
David Hu

Organization
Arm Limited

Contact
david.hu@arm.com

Introduction

This document proposes a generic prototype of the communication between NSPE (Non-secure Processing Environment) and SPE (Secure Processing Environment) in TF-M on a dual core system.

The dual core system has the following requirements

- NSPE and SPE are properly isolated and protected following PSA
- An Arm M-profile core locates in SPE and acts as the Secure core
- An Inter-Processor Communication hardware module in system for communication between NSPE core and SPE core

- TF-M runs on the Secure core with platform specific drivers support.

Scope

This design document focuses on the dual core communication design inside TF-M. Some changes to TF-M core/Secure Partition Manager (SPM) are listed to support the dual core communication. This document only discusses the implementation in TF-M Inter-Process Communication (IPC) model. The TF-M non-secure interface library depends on mailbox and NS RTOS implementation. The related changes to TF-M non-secure interface library are not discussed in detail in this document.

Some requirements to mailbox functionalities are defined in this document. The detailed mailbox design or implementations is not specified in this document. Please refer to mailbox dedicated document[1].

Organization of the document

- *Overall workflow in dual core communication* provides an overall workflow of dual core communication between NSPE and SPE.
- *Requirements on mailbox communication* lists general requirements of mailbox, from the perspective of dual core communication.
- *PSA client call handling flow in TF-M* discusses about the detailed sequence and key modules of handling PSA client call in TF-M.
- *Summary of changes to TF-M core/SPM* summarizes the potential changes to TF-M core/SPM to support dual core communication.

Overall workflow in dual core communication

The overall workflow in dual-core scenario can be described as follows

1. Non-secure application calls TF-M non-secure interface library to request Secure service. The TF-M non-secure interface library translates the Secure service into PSA Client calls.
2. TF-M non-secure interface library notifies TF-M of the PSA client call request, via mailbox. Proper generic mailbox APIs in HAL must be defined so that TF-M non-secure interface library can co-work with diverse platform specific Inter-Processor Communication implementations.
3. Inter-Processor Communication interrupt handler and mailbox handling in TF-M deal with the inbound mailbox event(s) and deliver the PSA client call request to TF-M SPM.
4. TF-M SPM processes the PSA client call request. The PSA client call is eventually handled in target Secure Partition or corresponding handler.
5. After the PSA Client call is completed, the return value is returned to NSPE via mailbox.
6. TF-M non-secure interface library fetches return value from mailbox.
7. The return value is returned to non-secure application.

The interfaces between NSPE app and TF-M NSPE interface library are unchanged so the underlying platform specific details are transparent to NSPE application.

Step 3 ~ step 5 are covered in *PSA client call handling flow in TF-M* in detail.

[1] *Mailbox Design in TF-M on Dual-core System*

Requirements on mailbox communication

The communication between NSPE and SPE relies on mailbox communication implementation. The mailbox functionalities are eventually implemented by platform specific Inter-Processor Communication drivers. This section lists some general requirements on mailbox communication between NSPE and SPE.

Data transferred between NPSE and SPE

A mailbox message shall contain the information and parameters of a PSA client call. After SPE is notified by a mailbox event, SPE fetches the parameters from NSPE for PSA Client call processing. The mailbox design document[Page 298, 1] defines the structure of the mailbox message.

The information and parameters of PSA Client call in the mailbox message include

- PSA Client API
- Parameters required in PSA Client call. The parameters can include the following, according to PSA client call type
  - Service ID (SID)
  - Handle
  - Request type
  - Input vectors and the lengths
  - Output vectors and the lengths
  - Requested version of secure service
- NSPE Client ID. Optional. The NSPE Client ID is required when NSPE RTOS enforces non-secure task isolation.

The mailbox implementation may define additional members in mailbox message to accomplish mailbox communication between NSPE and SPE.

When the PSA Client call is completed in TF-M, the return result, such as PSA_SUCCESS or a handle, shall be returned from SPE to NSPE via mailbox.

Mailbox synchronization between NSPE and SPE

Synchronization and protection between NSPE and SPE accesses to shared mailbox objects and variables shall be implemented.

When a core accesses shared mailbox objects or variables, proper mechanisms must protect concurrent operations from the other core.

Support of multiple ongoing NS PSA client calls (informative)

If the support of multiple ongoing NS PSA client calls in TF-M is required in dual-core systems, an optional queue can be maintained in TF-M core to store multiple mailbox objects received from NSPE. To identify NS PSA client calls, additional fields can be added in TF-M SPM objects to store the NS PSA Client request identification.

Note that when just a single outstanding PSA client call is allowed, multiple NSPE OS threads can run concurrently and call PSA client functions. The first PSA client call will be processed first, and any other OS threads will be blocked from submitting PSA client calls until the first is completed.

PSA client call handling flow in TF-M

This section provides more details about the flow of PSA client call handing in TF-M.

The sequence of handling PSA Client call request in TF-M is listed as below

1. Platform specific Inter-Processor Communication interrupt handler is triggered after the mailbox event is asserted by NSPE. The interrupt handler shall call `spm_handle_interrupt()`
2. SPM will send a `MAILBOX_INTERRUPT_SIGNAL` to `ns_agent_mailbox` partition
3. `ns_agent_mailbox` partition deals with the mailbox message(s) which contain(s) the PSA client call information and parameters. Then the PSA client call request is dispatched to dedicated PSA client call handler in TF-M SPM.
4. After the PSA client call is completed, the return value is transmitted to NSPE via mailbox.

Several key modules in the whole process are covered in detail in following sections.

- *Inter-Processor Communication interrupt handler* discusses the Inter-Processor Communication interrupt handler
- *TF-M Remote Procedure Call (RPC) layer* introduces TF-M Remote Procedure Call layer to support dual-core communication.
- *ns_agent_mailbox partition* describes the mailbox agent partition.
- *Return value replying routine in TF-M* proposes the routine to send the return value to NSPE.

Inter-Processor Communication interrupt handler

Platform specific driver shall implement the Inter-Processor Communication interrupt handler to deal with the Inter-Processor Communication interrupt asserted by NSPE. The platform specific interrupt handler shall complete the interrupt operations, such as interrupt EOI or acknowledge.

The interrupt handler shall call `spm_handle_interrupt()` to notify SPM of the interrupt.

The platform's `tfm_peripherals_def.h` file shall define a macro `MAILBOX_IRQ` that identifies the interrupt being used. The platform must also provide a function `mailbox_irq_init()` that initialises the interrupt as described in[2].

Platform specific driver shall put Inter-Processor Communication interrupt into a proper exception priority, according to system and application requirements. The proper priority setting must guarantee that

- TF-M can respond to a PSA client call request in time according to system and application requirements.
- Other exceptions, which are more latency sensitive or require higher priorities, are not blocked by Inter-Processor Communication interrupt ISR.

The exception priority setting is IMPLEMENTATION DEFINED.

[2] *Secure Interrupt Integration Guide*

TF-M Remote Procedure Call (RPC) layer

This design brings a concept of Remote Procedure Call layer into TF-M.

The RPC layer sits between TF-M SPM and mailbox implementation. The purpose of RPC layer is to decouple mailbox implementation and TF-M SPM and enhance the generality of entire dual-core communication.

The RPC layer provides a set of APIs to TF-M SPM to handle and reply PSA client call from NSPE in dual-core scenario. Please refer to *TF-M RPC definitions to TF-M SPM* for API details. It hides the details of specific mailbox implementation from TF-M SPM. It avoids modifying TF-M SPM to fit mailbox development and changes. It can keep a unified PSA client call process in TF-M SPM in both single Armv8-M scenario and dual core scenario.

The RPC layer defines a set callback functions for mailbox implementation to hook its specific mailbox operations. When TF-M SPM invokes RPC APIs to deal with NSPE PSA client call, RPC layer eventually calls the callbacks to execute mailbox operations. RPC layer also defines a set of PSA client call handler APIs for mailbox implementation. RPC specific client call handlers parse the PSA client call parameters and invoke common TF-M PSA client call handlers. Please refer to *TF-M RPC definitions for mailbox* for the details.

ns_agent_mailbox partition

A partition will be dedicated to interacting with the NSPE through the mailbox. This partition will call `tfm_hal_boot_ns_cpu()` and tfm_hal_wait_for_ns_cpu_ready() to ensure that the non-secure core is running. It will then initialise the SPE mailbox and enable the IPC interrupt. Once these tasks are complete, it will enter an infinite loop waiting for a `MAILBOX_INTERRUPT_SIGNAL` signal indicating that a mailbox message has arrived.

Mailbox handling will be done in the context of the `ns_agent_mailbox` partition, which will make any necessary calls to other partitions on behalf of the non-secure code.

`ns_agent_mailbox` shall call RPC API `tfm_rpc_client_call_handler()` to check and handle PSA client call request from NSPE. `tfm_rpc_client_call_handler()` invokes request handling callback function to eventually execute specific mailbox message handling operations. The mailbox APIs are defined in mailbox design document[Page 298, 1].

The handling process in mailbox operation consists of the following steps.

1. SPE mailbox fetches the PSA client call parameters from NSPE mailbox. Proper protection and synchronization must be implemented in mailbox to guarantee that the operations are not interfered with by NSPE mailbox operations or Inter-Processor Communication interrupt handler. If a queue is maintained inside TF-M core, SPE mailbox can fetch multiple PSA client calls together into the queue, to save the time of synchronization between two cores.
2. SPE mailbox parses the PSA client call parameters copied from NSPE, including the PSA client call type.
3. The PSA client call request is dispatched to the dedicated TF-M RPC PSA client call handler. The PSA client call request is processed in the corresponding handler.

- For `psa_framework_version()` and `psa_version()`, the PSA client call can be completed in the handlers `tfm_rpc_psa_framework_version()` and `tfm_rpc_psa_version()` respectively.
- For `psa_connect()`, `psa_call()` and `psa_close()`, the handlers `tfm_rpc_psa_connect()`, `tfm_rpc_psa_call()` and `tfm_rpc_psa_close()` create the PSA message and trigger target Secure partition respectively. The target Secure partition will be woken up to handle the PSA message.

The dual-core scenario and single Armv8-M scenario in TF-M IPC implementation will share the same PSA client call routines inside TF-M SPM. The current handler definitions can be adjusted to be more generic for dual-core scenario and single Armv8-M implementation. Please refer to *Summary of changes to TF-M core/SPM* for details.

If there are multiple NSPE PSA client call requests pending, SPE mailbox can process mailbox messages one by one.

Return value replying routine in TF-M

Diverse PSA client calls can be implemented with different return value replying routines.

- *Replying routine for psa_framework_version() and psa_version()* describes the routine for `psa_framework_version()` and `psa_version()`.
- *Replying routine for psa_connect(), psa_call() and psa_close()* describes the routine for `psa_connect()`, `psa_call()` and `psa_close()`.

Replying routine for psa_framework_version() and psa_version()

For `psa_framework_version()` and `psa_version()`, the return value can be directly returned from the dedicated TF-M RPC PSA client call handlers. Therefore, the return value can be directly replied in mailbox handling process.

A compile flag may be defined to enable replying routine via mailbox in dual-core scenario during building.

Replying routine for psa_connect(), psa_call() and psa_close()

For `psa_connect()`, `psa_call()` and `psa_close()`, the PSA client call is completed in the target Secure Partition. The target Secure Partition calls `psa_reply()` to reply the return value to TF-M SPM. In the SVC handler of `psa_reply()` in TF-M SPM, TF-M SPM should call TF-M RPC API `tfm_rpc_client_call_reply()` to return the value to NSPE via mailbox. `tfm_rpc_client_call_reply()` invokes reply callbacks to execute specific mailbox reply operations. The mailbox reply functions must not trigger a context switch inside SVC handler.

If an error occurs in the handlers, the TF-M RPC handlers, `tfm_rpc_psa_call()`, `tfm_rpc_psa_connect()` and `tfm_rpc_psa_close()`, may terminate and return the error, without triggering the target Secure Partition. The mailbox implementation shall return the error code to NSPE.

Summary of changes to TF-M core/SPM

This section discusses the general changes related to NSPE and SPE communication to current TF-M core/SPM implementations.

The detailed mailbox implementations are not covered in this section. Please refer to mailbox dedicated document[Page 298, 1]. The platform specific implementations are also not covered in this section, including the Inter-Processor Communication interrupt or its interrupt handler.

Common PSA client call handlers

Common PSA client call handlers shall be extracted from current PSA client call handlers implementation in TF-M. Common PSA client call handlers are shared by both TF-M RPC layer in dual-core scenario and SVCall handlers in single Armv8-M scenario.

TF-M RPC layer

This section describes the TF-M RPC data types and APIs.

- *TF-M RPC definitions to TF-M SPM* lists the data types and APIs to be invoked by TF-M SPM.
- *TF-M RPC definitions for mailbox* lists the data types and APIs to be referred by mailbox implementation

TF-M RPC definitions to TF-M SPM

TFM_RPC_SUCCESS

TFM_RPC_SUCCESS is a general return value to indicate that the RPC operation succeeds.

```
#define TFM_RPC_SUCCESS             (0)
```

TFM_RPC_INVAL_PARAM

TFM_RPC_INVAL_PARAM is a return value to indicate that the input parameters are invalid.

```
#define TFM_RPC_INVAL_PARAM         (INT32_MIN + 1)
```

TFM_RPC_CONFLICT_CALLBACK

Currently one and only one mailbox implementation is supported in dual core communication. This flag indicates that callback functions from one mailbox implementation are already registered and no more implementations are accepted.

```
#define TFM_RPC_CONFLICT_CALLBACK   (INT32_MIN + 2)
```

tfm_rpc_client_call_handler()

TF-M PendSV handler calls this function to handle NSPE PSA client call request.

```
void tfm_rpc_client_call_handler(void);
```

Usage

tfm_rpc_client_call_handler() invokes callback function handle_req() to execute specific mailbox handling. Please note that tfm_rpc_client_call_handler() doesn't return the status of underlying mailbox handling.

tfm_rpc_client_call_reply()

TF-M psa_reply() handler calls this function to reply PSA client call return result to NSPE.

```
void tfm_rpc_client_call_reply(const void *owner, int32_t ret);
```

Parameters

owner	A handle to identify the owner of the PSA client call return value.
ret	PSA client call return result value.

Usage

tfm_rpc_client_call_reply() invokes callback function reply() to execute specific mailbox reply. Please note that tfm_rpc_client_call_reply() doesn't return the status of underlying mailbox reply process.

TF-M RPC definitions for mailbox

Mailbox operations callbacks

This structures contains the callback functions for specific mailbox operations.

```
struct tfm_rpc_ops_t {
    void (*handle_req)(void);
    void (*reply)(const void *owner, int32_t ret);
};
```

tfm_rpc_register_ops()

This function registers underlying mailbox operations into TF-M RPC callbacks.

```
int32_t tfm_rpc_register_ops(const struct tfm_rpc_ops_t *ops_ptr);
```

Parameters

ops_ptr	Pointer to the specific operation structure.

Return

TFM_RPC_SUCCESS	Operations are successfully registered.
Other error code	Fail to register operations.

Usage

Mailbox shall register TF-M RPC callbacks during mailbox initialization, before enabling secure services for NSPE.

Currently one and only one underlying mailbox communication implementation is allowed in runtime.

tfm_rpc_unregister_ops()

This function unregisters underlying mailbox operations from TF-M RPC callbacks.

```
void tfm_rpc_unregister_ops(void);
```

Usage

Currently one and only one underlying mailbox communication implementation is allowed in runtime.

tfm_rpc_psa_framework_version()

TF-M RPC handler for psa_framework_version().

```
uint32_t tfm_rpc_psa_framework_version(void);
```

Return

version	The version of the PSA Framework implementation that is providing the runtime services.

Usage

`tfm_rpc_psa_framework_version()` invokes common `psa_framework_version()` handler in TF-M.

tfm_rpc_psa_version()

TF-M RPC handler for psa_version().

```
uint32_t tfm_rpc_psa_version(uint32_t sid);
```

Parameters

sid	RoT Service identity.

Return

PSA_VERSION_NONE	The RoT Service is not implemented, or the caller is not permitted to access the service.
> 0	The minor version of the implemented RoT Service.

Usage

`tfm_rpc_psa_version()` invokes common `psa_version()` handler in TF-M. The parameters in params shall be prepared before calling `tfm_rpc_psa_version()`.

tfm_rpc_psa_connect()

TF-M RPC handler for psa_connect().

```
psa_status_t tfm_rpc_psa_connect(uint32_t sid,
                                 uint32_t version,
                                 int32_t ns_client_id,
                                 const void *client_data);
```

Parameters

sid	RoT Service identity.
version	The version of the RoT Service.
ns_client_id	Agent representing NS client's identifier.
client_data	Client data, treated as opaque by SPM.

Return

PSA_SUCCESS	Success.
PSA_CONNECTION_BUSY	The SPM cannot make the connection at the moment.
Does not return	The RoT Service ID and version are not supported, or the caller is not permitted to access the service.

Usage

tfm_rpc_psa_connect() invokes common psa_connect() handler in TF-M.

tfm_rpc_psa_call()

TF-M RPC handler for psa_call().

```
psa_status_t tfm_rpc_psa_call(psa_handle_t handle, uint32_t control,
                              const struct client_params_t *params,
                              const void *client_data_stateless);
```

Parameters

handle	Handle to the service being accessed.
control	A composited uint32_t value for controlling purpose, containing call types, numbers of in/out vectors and attributes of vectors.
params	Combines the psa_invec and psa_outvec params for the psa_call() to be made, as well as NS agent's client identifier, which is ignored for connection-based services.
client_data_stateless	Client data, treated as opaque by SPM.

Return

PSA_SUCCESS	Success.
Does not return	The call is invalid, or invalid parameters.

Usage

`tfm_rpc_psa_call()` invokes common `psa_call()` handler in TF-M. The value of control and parameters in params shall be prepared before calling `tfm_rpc_psa_call()`.

`tfm_rpc_psa_close()`

TF-M RPC `psa_close()` handler

```
psa_status_t tfm_rpc_psa_close(psa_handle_t handle);
```

Parameters

`handle`	A handle to an established connection.

Return

`PSA_SUCCESS`	Success.
`PROGRAMMER ERROR`	The call is invalid, or invalid parameters.

Usage

`tfm_rpc_psa_close()` invokes common `psa_close()` handler in TF-M.

Other modifications

The following mandatory changes are also required.

- One or more compile flag(s) shall be defined to select corresponding execution routines in dual-core scenario or single Armv8-M scenario during building.

Reference

11.2.3 Mailbox Design in TF-M on Dual-core System

Author
David Hu

Organization
Arm Limited

Contact
david.hu@arm.com

Introduction

This document proposes a generic design of the mailbox communication for Trusted Firmware-M (TF-M) on a dual-core system. The mailbox functionalities transfer PSA Client requests and results between Non-secure Processing Environment (NSPE) and Secure Processing Environment (SPE).

The dual-core system should satisfy the following requirements

- NSPE and SPE are properly isolated and protected following PSA specifications.
- An Arm M-profile core locates in SPE and acts as the Secure core.
- TF-M runs on the Secure core with platform specific drivers support.
- Inter-Processor Communication hardware module in system for communication between Secure core and Non-secure core. Mailbox communication calls the Inter-Processor Communication to transfer notifications.
- Non-secure memory shared by NSPE and SPE.

Scope

This design document focuses on the mailbox functionalities in NSPE and SPE on a dual-core system. The mailbox functionalities include the initialization of mailbox in NSPE and SPE, and the transfer of PSA Client requests and replies between NSPE and SPE.

Data types and mailbox APIs are defined in this document.

Some details of interactions between mailbox and other modules are specified in other documents. Communication prototype design[1] defines a general communication prototype between NSPE and SPE on a dual-core system. It describes how TF-M interacts with mailbox functionalities and the general requirements on mailbox. Dual-core booting sequence[2] describes a synchronization step of mailbox between NSPE and SPE during system booting.

Organization of the document

- *Mailbox architecture* provides an overview on the mailbox architecture.
- *Mailbox communication for PSA Client calls* discusses about the mailbox communication for PSA Client calls.
- *Mailbox initialization* introduces the initialization of mailbox.
- *Mailbox APIs and data structures* lists mailbox data types and APIs.

Mailbox architecture

The mailbox consists of two parts sitting in NSPE and SPE respectively. NSPE mailbox provides mailbox functionalities in NSPE and SPE mailbox provides mailbox functionalities in TF-M in SPE.

PSA Client APIs called in NSPE are implemented by NSPE mailbox functions on dual-core systems, to send PSA Client request and receive the result. The implementation can vary in diverse NSPE OSs or use cases.

TF-M provides a reference implementation of NSPE mailbox. The NSPE mailbox delivers the PSA Client requests to SPE mailbox. After the PSA Client result is replied from SPE, NSPE mailbox fetches the result and returns it to PSA Client APIs.

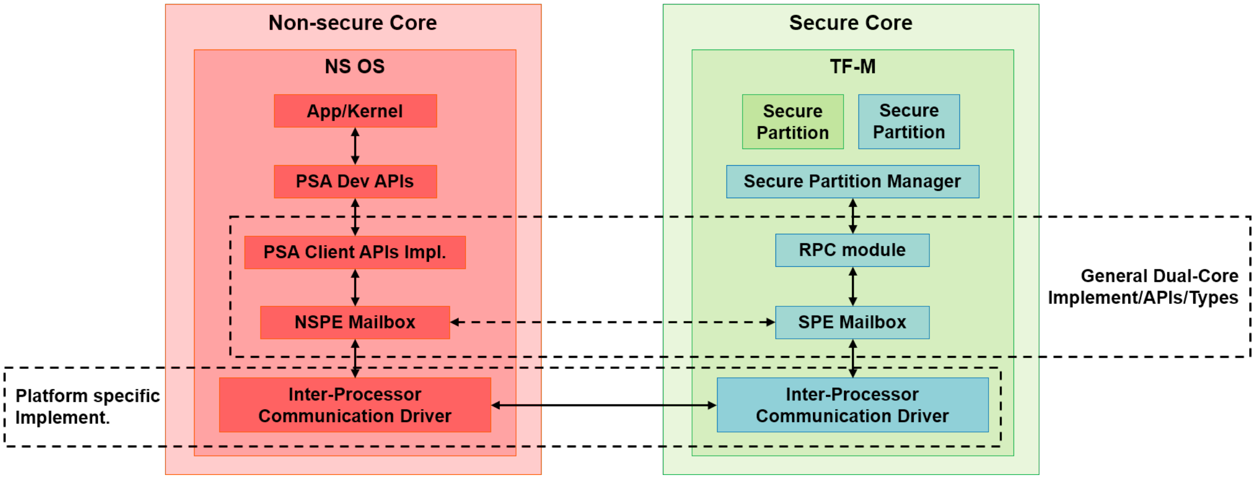
[1] *Communication prototype between NSPE and SPE in Dual-core systems*

[2] *Booting a Dual-core system*

NSPE mailbox objects are managed by NSPE mailbox in non-secure memory to hold PSA Client call parameters, return result and other mailbox data. NSPE mailbox relies on platform specific Inter-Process Communication to process notifications between NSPE and SPE.

The SPE mailbox in TF-M receives PSA Client requests from NSPE mailbox. It parses the requests and invokes TF-M Remote Procedure Call (RPC) layer. RPC layer delivers the requests to TF-M core/Secure Partition Manager (SPM). After the PSA Client call is completed, TF-M core/SPM invokes RPC layer to return results to NSPE, via SPE mailbox. SPE mailbox objects are managed by SPE mailbox in secure memory. SPE mailbox relies on platform specific Inter-Process Communication to process notifications between SPE and NSPE.

The architecture is showed in following figure.



Mailbox communication for PSA Client calls

This section describes the transfer of PSA Client request and reply between NSPE and SPE via mailbox.

Mailbox objects

This section lists the mailbox objects required in NSPE and SPE.

NSPE mailbox objects are managed by NSPE mailbox in non-secure memory. But NSPE mailbox objects can be accessed by both NSPE mailbox and SPE mailbox.

SPE mailbox objects are managed by SPE mailbox in secure memory. SPE mailbox objects should be protected from NSPE accesses by system specific isolation.

NSPE Mailbox queue

NSPE mailbox maintains a mailbox queue in non-secure memory. Please refer to the structure definition in *NSPE mailbox queue structure*.

NSPE mailbox queue contains one or more slots. The number of slots should be aligned with that in SPE mailbox queue.

Each slot in NSPE mailbox queue consists of a pair of a mailbox message structure and a mailbox reply structure. Each slot might contain additional fields, such as identification of non-secure task which initiates the PSA Client request. Each slot serves a PSA Client call from non-secure task.

The parameters of PSA Client request are hosted in the mailbox message inside the slot. *Mailbox messages* describes the details of mailbox message.

The mailbox reply structure is used to receive the PSA Client result from SPE. *Mailbox replies* describes the details of mailbox reply.

Mailbox messages

A mailbox message contains the parameters of a PSA Client request from a non-secure task. Please refer to the structure definition in *Mailbox message structure*.

Inside PSA Client API implementation, NSPE mailbox selects an empty mailbox queue slot for the PSA Client request. The parameters of that PSA Client request are organized into the mailbox message belonging to the selected slot. SPE mailbox will parse those parameters from the mailbox message.

More fields can be defined in mailbox message to transfer additional information from NSPE to SPE for processing in TF-M.

Mailbox replies

A mailbox reply structure in non-secure memory receives the PSA Client result replied from SPE mailbox. Please refer to the structure definition in *Mailbox reply structure*.

SPE Mailbox queue

SPE mailbox maintains a mailbox queue to store SPE mailbox objects. Please refer to the structure definition in *SPE mailbox queue structure*.

SPE mailbox queue contains one or more slots. The number of slots should be aligned with that in NSPE mailbox queue. After SPE is notified that a PSA Client request is pending, SPE mailbox can assign an empty slot, copy the corresponding PSA Client call parameters from non-secure memory to that slot and parse the parameters.

Each slot in SPE mailbox queue can contain the following fields

- An optional field to hold mailbox message content copied from non-secure memory.
- Index of NSPE mailbox queue slot containing the mailbox message.
- A handle to the mailbox message. Optional. Identify the owner slot of PSA Client result when multiple mailbox messages are under processing.

More fields can be defined in the slot structure to support mailbox processing in SPE.

Overall workflow

The overall workflow of transferring PSA Client requests and results between NSPE and SPE via mailbox is shown below.

1. Non-secure task initiates a service request by calling PSA Developer APIs, which eventually invoke PSA Client APIs. PSA Client APIs call NSPE mailbox functions to transmit PSA Client call to SPE.
2. NSPE mailbox assigns an empty slot from NSPE mailbox queue for that PSA client call.
3. NSPE mailbox prepares the parameters of PSA Client call in the dedicated mailbox message inside the assigned slot.

4. After the mailbox message is ready, NSPE mailbox invokes platform specific Inter-Processor Communication driver to notify SPE. The notification mechanism of Inter-Processor Communication is platform specific.
5. After the notification is completed, non-secure task waits for the reply from SPE.
6. Platform specific Inter-Processor Communication interrupt for mailbox is asserted in SPE. The interrupt handler activates SPE mailbox to process the request(s).
7. During mailbox processing in TF-M, the handling routine can include the following steps:
    1. SPE mailbox checks and validates NSPE mailbox queue status.
    2. SPE mailbox fetches PSA Client call parameters from NSPE mailbox queue.
    3. SPE mailbox parses the parameters.
    4. SPE mailbox invokes the TF-M RPC APIs to deliver the PSA Client request to TF-M SPM.
    5. The PSA Client call is handled in TF-M SPM and target Secure Partition if necessary.

    If multiple ongoing mailbox messages are pending in the SPE, SPE mailbox can process mailbox messages one by one.
8. After the PSA Client call is completed, TF-M RPC layer notifies SPE mailbox to reply PSA Client result to NSPE.
9. SPE mailbox writes the PSA Client result to the dedicated mailbox reply structure in non-secure memory. The related SPE mailbox objects should be invalidated or cleaned.
10. SPE mailbox notifies NSPE by invoking Inter-Processor Communication driver to send a notification to NSPE. The notification mechanism of Inter-Processor Communication is platform specific.
11. NSPE mailbox is activated to handle the PSA Client result in the mailbox reply structure. Related mailbox objects should be invalidated or cleaned by NSPE mailbox after the return results is extracted out.
12. NSPE mailbox returns the result to PSA Client API implementation. The result is eventually returned to the non-secure task.

The following sections discuss more details of key steps in above sequence.

Mailbox notifications between NSPE and SPE

As shown in *Overall workflow*, NSPE mailbox asserts mailbox notification to trigger SPE to handle PSA Client request. SPE mailbox asserts mailbox notification to notify NSPE that PSA Client result is written. The notification implementation is based on platform specific Inter-Processor Communication.

It is recommended to assign one independent set of Inter-Processor Communication channel to each notification routine respectively, to implement a *full-duplex* notification mechanism between NSPE and SPE. If both notification routines share the same Inter-Processor Communication channel, proper synchronization should be implemented to prevent conflicts between two notification routines.

In SPE, the Inter-Processor Communication interrupt handler should deal with the incoming notification from NSPE and activate the subsequent mailbox handling in SPE. Communication prototype design[Page 308, 1] defines the behavior of Inter-Processor Communication interrupt handler.

NSPE can implement an interrupt handler or a polling of notification status to handle Inter-Processor Communication notification from SPE.

Implement PSA Client API with NSPE Mailbox

PSA Client APIs are implemented with NSPE mailbox API `tfm_ns_mailbox_client_call()`.

The pseudo code below shows a reference implementation of `psa_framework_version()`.

```
uint32_t psa_framework_version(void)
{
    ...
    int32_t ret;

    ret = tfm_ns_mailbox_client_call(...);
    if (ret != MAILBOX_SUCCESS) {
        version = PSA_VERSION_NONE;
    }

    ...
}
```

`tfm_ns_mailbox_client_call()` implementation can vary according to usage scenario. TF-M reference implementation provides implementations for NS OS and NS bare metal environment respectively. Refer to *TF-M reference implementation of NSPE mailbox* for details.

As PSA Firmware Framework-M (FF-M) requests, a PSA Client API function should be blocked until the result is returned. To comply with FF-M, NSPE mailbox requires proper mechanism(s) to keep current caller waiting for PSA Client result or an empty mailbox queue slot.

Note: `tfm_ns_mailbox_client_call()` may trap the current exception in sleep and therefore it must not be called in interrupt service routine.

Refer to *Mailbox APIs and data structures* for details of `tfm_ns_mailbox_client_call()`.

TF-M reference implementation of NSPE mailbox

TF-M NS interface provides a reference implementation of NS mailbox.

This reference implementation defines several NS mailbox HAL APIs. Please refer to *NSPE mailbox HAL APIs* for details.

Integration with NSPE

TF-M reference implementation provides several mailbox build flags to control the integration with NS software.

- `TFM_MULTI_CORE_NS_OS`

  When integrating NS mailbox with NS OS, such as NS RTOS, that flag can be selected to enable NS OS support in NS mailbox, such as thread management to fulfill thread wait and wake-up. Please refer to *NSPE mailbox RTOS abstraction APIs* for NS OS support details.

  With NS OS support, multiple outstanding PSA Client calls can be supported in NS mailbox when number of mailbox queue slots configured in `NUM_MAILBOX_QUEUE_SLOT` is greater than 1.

  If `TFM_MULTI_CORE_NS_OS` is enabled, when a NS client starts a PSA Client call:

    - `tfm_ns_mailbox_client_call()` selects an empty NSPE mailbox queue slot to organize received PSA client call parameters into a mailbox message.
    - Then it sends those parameters to SPE mailbox and waits for results from SPE. During waiting for the result, the NS client thread may be switched out by NS OS scheduler.
    - When the result arrives, the NS client thread will be woken up inside NS mailbox interrupt handler.
    - The result is then written back to NS client finally.

    When that flag is disabled, NS mailbox runs as being integrated with NS bare metal environment. NS mailbox simply loops mailbox message status while waiting for results.

- `TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD`

  When `TFM_MULTI_CORE_NS_OS` is enabled, this flag can be selected to enable another NS mailbox thread model which relies on a NS mailbox dedicated thread.

  - It requires NS OS to create a dedicated thread to perform NS mailbox functionalities. This dedicated thread invokes `tfm_ns_mailbox_thread_runner()` to handle PSA Client calls. `tfm_ns_mailbox_thread_runner()` constructs mailbox messages and sends them to SPE mailbox.
  - `tfm_ns_mailbox_client_call()` sends PSA Client calls to the dedicated mailbox thread. It doesn't directly deal with mailbox messages.
  - It also relies on NS OS to provide thread management and inter-thread communication. Please refer to *NSPE mailbox RTOS abstraction APIs* for details.
  - It also requires dual-cpu platform to implement NS Inter-Processor Communication interrupts. The interrupt handler invokes `tfm_ns_mailbox_wake_reply_owner_isr()` to deal with PSA Client call replies and notify the waiting threads.

Multiple outstanding PSA Client call feature

Multiple outstanding PSA Client call feature can enable dual-cpu platform to issue multiple PSA Client calls in NS OS and those PSA Client calls can be served simultaneously.

Without this feature, only a single PSA Client call can be issued and served. A new PSA Client call cannot be started until the previous one is completed.

When multiple outstanding PSA Client call feature is enabled, while a NS application is waiting for its PSA Client result, another NS application can be switched in by NS OS to prepare another PSA Client call or deal with its PSA client result. It can decrease the CPU idle time of waiting for PSA Client call completion.

If multiple NS applications request secure services in NS OS, it is recommended to enable this feature.

To implement this feature in NS OS:

- Platform should set the number of mailbox queue slots in `NUM_MAILBOX_QUEUE_SLOT` in platform's `config.cmake`. It will use more data area with multiple mailbox queue slots.

  NSPE and SPE share the same `NUM_MAILBOX_QUEUE_SLOT` value.

- Enable `TFM_MULTI_CORE_NS_OS`

  For more details, refer to *TFM_MULTI_CORE_NS_OS*.

TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD can be enabled to select another NS mailbox working model. See *TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD* for details.

Critical section protection between cores

Proper protection should be implemented to protect the critical accesses to shared mailbox resources. The critical sections can include atomic reading and modifying NSPE mailbox queue status, slot status and other critical operations.

The implementation should protect a critical access to those shared mailbox resource from corruptions caused by accesses from the peer core. SPE mailbox also accesses NSPE mailbox queue. Therefore, it is essential to implement synchronization or protection on NSPE mailbox queue between Secure core and Non-secure core. NSPE mailbox and SPE mailbox define corresponding critical section protection APIs respectively. The implementation of those APIs can be platform specific. Please see more details in *NSPE mailbox APIs* and *SPE mailbox APIs*.

It is recommended to rely on both hardware and software to implement the synchronization and protection.

Protection of local mailbox objects can be implemented as static functions inside NSPE mailbox and SPE mailbox.

Mailbox handling in TF-M

According to communication prototype design[Page 308, 1], mailbox implementation should invoke tfm_rpc_register_ops() to hook its operations to TF-M RPC module callbacks during initialization. Mailbox message handling should call TF-M RPC PSA Client call handlers to deliver PSA Client request to TF-M SPM.

If multiple outstanding NS PSA Client calls should be supported, TF-M SPM can store the mailbox message handle in a specific field in PSA message structure to identify the mailbox message, while creating a PSA message. While replying the PSA Client result, TF-M SPM can extract the mailbox message handle from PSA message and pass it back to mailbox reply function. SPE mailbox can identify which mailbox message is completed according to the handle and write the result to corresponding NSPE mailbox queue slot.

Platform specific Inter-Processor Communication interrupt handler in SPE should call spm_handle_interrupt() to notify SPM of the interrupt. SPM will then send the MAILBOX_INTERRUPT_SIGNAL signal to the ns_agent_mailbox partition, which will call tfm_rpc_client_call_handler().

Mailbox initialization

It should be guaranteed that NSPE mailbox should not initiate PSA Client request until SPE mailbox initialization completes. Refer to dual-core booting sequence[Page 308, 2] for more details on the synchronization between NSPE and SPE during booting.

In current design, the base address of NSPE mailbox queue should be pre-defined and shared between NSPE mailbox and SPE mailbox.

SPE mailbox initialization

The SPE mailbox queue memory should be allocated before calling `tfm_mailbox_init()`. `tfm_mailbox_init()` initializes the memory and variables. `tfm_mailbox_init()` calls `tfm_mailbox_hal_init()` to perform platform specific initialization. The base address of NSPE mailbox queue can be received via `tfm_mailbox_hal_init()`.

SPE mailbox dedicated Inter-Processor Communication initialization can also be enabled during SPE mailbox initialization.

After SPE mailbox initialization completes, SPE notifies NSPE that SPE mailbox functionalities are ready.

NSPE mailbox initialization

The NSPE mailbox queue memory should be allocated before calling `tfm_ns_mailbox_init()`. `tfm_ns_mailbox_init()` initializes the memory and variables. `tfm_ns_mailbox_init()` calls `tfm_ns_mailbox_hal_init()` to perform platform specific initialization. The base address of NSPE mailbox queue can be passed to SPE mailbox via `tfm_ns_mailbox_hal_init()`.

NSPE mailbox dedicated Inter-Processor Communication initialization can also be enabled during NSPE mailbox initialization.

Mailbox APIs and data structures

Data types

Constants

MAILBOX_SUCCESS

`MAILBOX_SUCCESS` is a generic return value to indicate success of mailbox operation.

```
#define MAILBOX_SUCCESS        (0)
```

MAILBOX_QUEUE_FULL

`MAILBOX_QUEUE_FULL` is a return value from mailbox function if mailbox queue is full.

```
#define MAILBOX_QUEUE_FULL     (INT32_MIN + 1)
```

MAILBOX_INVAL_PARAMS

`MAILBOX_INVAL_PARAMS` is a return value from mailbox function if any parameter is invalid.

```
#define MAILBOX_INVAL_PARAMS   (INT32_MIN + 2)
```

MAILBOX_NO_PERMS

MAILBOX_NO_PERMS is a return value from mailbox function if the caller doesn't own a proper permission to execute the operation.

```
#define MAILBOX_NO_PERMS       (INT32_MIN + 3)
```

MAILBOX_NO_PEND_EVENT

MAILBOX_NO_PEND_EVENT is a return value from mailbox function if the expected event doesn't occur yet.

```
#define MAILBOX_NO_PEND_EVENT  (INT32_MIN + 4)
```

MAILBOX_CHAN_BUSY

MAILBOX_CHAN_BUSY is a return value from mailbox function if the underlying Inter-Processor Communication resource is busy.

```
#define MAILBOX_CHAN_BUSY      (INT32_MIN + 5)
```

MAILBOX_CALLBACK_REG_ERROR

MAILBOX_CALLBACK_REG_ERROR is a return value from mailbox function if the registration of mailbox callback functions failed.

```
#define MAILBOX_CALLBACK_REG_ERROR     (INT32_MIN + 6)
```

MAILBOX_INIT_ERROR

MAILBOX_INIT_ERROR is a return value from mailbox function if the mailbox initialization failed.

```
#define MAILBOX_INIT_ERROR     (INT32_MIN + 7)
```

MAILBOX_GENERIC_ERROR

MAILBOX_GENERIC_ERROR indicates mailbox generic errors which cannot be indicated by the codes above.

```
#define MAILBOX_GENERIC_ERROR    (INT32_MIN + 8)
```

PSA Client API types

The following constants define the PSA Client API type values shared between NSPE and SPE

```
#define MAILBOX_PSA_FRAMEWORK_VERSION       (0x1)
#define MAILBOX_PSA_VERSION                 (0x2)
#define MAILBOX_PSA_CONNECT                 (0x3)
#define MAILBOX_PSA_CALL                    (0x4)
#define MAILBOX_PSA_CLOSE                   (0x5)
```

Mailbox message structure

psa_client_params_t lists the parameters passed from NSPE to SPE required by a PSA Client call.

```
struct psa_client_params_t {
    union {
        struct {
            uint32_t        sid;
        } psa_version_params;

        struct {
            uint32_t        sid;
            uint32_t        minor_version;
        } psa_connect_params;

        struct {
            psa_handle_t    handle;
            int32_t         type;
            const psa_invec *in_vec;
            size_t          in_len;
            psa_outvec      *out_vec;
            size_t          out_len;
        } psa_call_params;

        struct {
            psa_handle_t    handle;
        } psa_close_params;
    };
};
```

The following structure describe a mailbox message and its members.

- call_type indicates the PSA Client API type.
- params stores the PSA Client call parameters.
- client_id records the client ID of the non-secure client. Optional. It is used to identify the non-secure tasks in TF-M when NSPE OS enforces non-secure task isolation.

```
struct mailbox_msg_t {
    uint32_t                     call_type;
    struct psa_client_params_t   params;

```

(continues on next page)

(continued from previous page)

```
    int32_t                      client_id;
};
```

Mailbox reply structure

This structure describes a mailbox reply structure, which is managed by NSPE mailbox in non-secure memory.

```
struct mailbox_reply_t {
    int32_t    return_val;
    const void *owner;
    int32_t    *reply;
    uint8_t    *woken_flag;
};
```

Mailbox queue status bitmask

mailbox_queue_status_t defines a bitmask to indicate a status of slots in mailbox queues.

```
typedef uint32_t   mailbox_queue_status_t;
```

NSPE mailbox queue structure

ns_mailbox_slot_t defines a non-secure mailbox queue slot.

```
/* A single slot structure in NSPE mailbox queue */
struct ns_mailbox_slot_t {
    struct mailbox_msg_t   msg;
    struct mailbox_reply_t reply;
};
```

ns_mailbox_queue_t describes the NSPE mailbox queue and its members in non-secure memory.

- empty_slots is the bitmask of empty slots.
- pend_slots is the bitmask of slots whose PSA Client call is not replied yet.
- replied_slots is the bitmask of slots whose PSA Client result is returned but not extracted yet.
- queue is the NSPE mailbox queue of slots.
- is_full indicates whether NS mailbox queue is full.

```
struct ns_mailbox_queue_t {
    mailbox_queue_status_t   empty_slots;
    mailbox_queue_status_t   pend_slots;
    mailbox_queue_status_t   replied_slots;

    struct ns_mailbox_slot_t queue[NUM_MAILBOX_QUEUE_SLOT];

    bool                     is_full;
};
```

SPE mailbox queue structure

secure_mailbox_slot_t defines a single slot structure in SPE mailbox queue.

- ns_slot_idx records the index of NSPE mailbox slot containing the mailbox message under processing. SPE mailbox determines the reply structure address according to this index.
- msg_handle contains the handle to the mailbox message under processing. The handle can be delivered to TF-M SPM while creating PSA message to identify the mailbox message.

```
/* A handle to a mailbox message in use */
typedef int32_t    mailbox_msg_handle_t;

struct secure_mailbox_slot_t {
    uint8_t              ns_slot_idx;
    mailbox_msg_handle_t msg_handle;
};
```

secure_mailbox_queue_t describes the SPE mailbox queue in secure memory.

- empty_slots is the bitmask of empty slots.
- queue is the SPE mailbox queue of slots.
- ns_queue stores the address of NSPE mailbox queue structure.

```
struct secure_mailbox_queue_t {
    mailbox_queue_status_t       empty_slots;

    struct secure_mailbox_slot_t queue[NUM_MAILBOX_QUEUE_SLOT];
    /* Base address of NSPE mailbox queue in non-secure memory */
    struct ns_mailbox_queue_t    *ns_queue;
};
```

NSPE mailbox APIs

NSPE mailbox interface APIs

APIs defined in this section are called by NS software and PSA Client APIs implementations.

tfm_ns_mailbox_init()

This function initializes NSPE mailbox.

```
int32_t tfm_ns_mailbox_init(struct ns_mailbox_queue_t *queue);
```

Parameters

queue	The base address of NSPE mailbox queue.

Return

MAILBOX_SUCCESS	Initialization succeeds.
Other return codes	Initialization fails with an error code.

Usage

tfm_ns_mailbox_init() invokes tfm_ns_mailbox_hal_init() to complete platform specific mailbox and Inter-Processor Communication initialization. The non-secure memory area for NSPE mailbox queue structure should be statically or dynamically pre-allocated before calling tfm_ns_mailbox_init().

tfm_ns_mailbox_client_call()

This function sends the PSA Client request to SPE, waits and fetches PSA Client result.

```
int32_t tfm_ns_mailbox_client_call(uint32_t call_type,
                                   const struct psa_client_params_t *params,
                                   int32_t client_id,
                                   int32_t *reply);
```

Parameters

call_type	Type of PSA Client call
params	Address of PSA Client call parameters structure.
client_id	ID of non-secure task.
reply	The NS client task private buffer written with PSA Client result

Return

MAILBOX_SUCCESS	PSA Client call is completed successfully.
Other return code	Operation failed with an error code.

Usage

If TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is enabled, tfm_ns_mailbox_client_call() will forward PSA Client calls to the dedicated mailbox thread via NS OS message queue. Otherwise, tfm_ns_mailbox_client_call() directly deals with PSA Client calls and perform NS mailbox functionalities.

tfm_ns_mailbox_thread_runner()

This function handles PSA Client call inside a dedicated NS mailbox thread. It constructs mailbox messages and transmits them to SPE mailbox.

```
void tfm_ns_mailbox_thread_runner(void *args);
```

Parameters

args	The pointer to the structure of PSA Client call parameters.

Usage

tfm_ns_mailbox_thread_runner() should be executed inside the dedicated mailbox thread.

Note: `tfm_ns_mailbox_thread_runner()` is implemented as an empty function when `TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD` is disabled.

`tfm_ns_mailbox_wake_reply_owner_isr()`

This function wakes up the owner task(s) of the returned PSA Client result(s).

```
int32_t tfm_ns_mailbox_wake_reply_owner_isr(void);
```

Return

`MAILBOX_SUCCESS`	The tasks of replied mailbox messages were found and wake-up signals were sent.
`MAILBOX_NO_PEND_EVENT`	No replied mailbox message is found.
Other return codes	Operation failed with an error code

Usage

`tfm_ns_mailbox_wake_reply_owner_isr()` should be called from platform specific Inter-Processor Communication interrupt handler.

Note: `tfm_ns_mailbox_wake_reply_owner_isr()` is implemented as a dummy function when `TFM_MULTI_CORE_NS_OS` is disabled.

NSPE mailbox HAL APIs

The HAL APIs defined in this section should be implemented by platform-specific implementation.

This section describes a *reference design* of NSPE mailbox HAL APIs. Developers can define and implement different APIs.

`tfm_ns_mailbox_hal_init()`

This function executes platform-specific NSPE mailbox initialization.

```
int32_t tfm_ns_mailbox_hal_init(struct ns_mailbox_queue_t *queue);
```

Parameters

queue	The base address of NSPE mailbox queue.

Return

`MAILBOX_SUCCESS`	Initialization succeeds.
Other return codes	Initialization fails with an error code.

Usage

tfm_ns_mailbox_hal_init() performs platform specific mailbox and Inter-Processor Communication initialization. tfm_ns_mailbox_hal_init() can also share the address of NSPE mailbox queue with SPE mailbox via platform specific implementation.

tfm_ns_mailbox_hal_notify_peer()

This function invokes platform specific Inter-Processor Communication drivers to send notification to SPE.

```
int32_t tfm_ns_mailbox_hal_notify_peer(void);
```

Return

MAILBOX_SUCCESS	The operation completes successfully.
Other return codes	Operation fails with an error code.

Usage

tfm_ns_mailbox_hal_notify_peer() should be implemented by platform specific Inter-Processor Communication drivers.

tfm_ns_mailbox_hal_notify_peer() should not be exported outside NSPE mailbox.

tfm_ns_mailbox_hal_enter_critical()

This function enters the critical section of NSPE mailbox queue access.

```
void tfm_ns_mailbox_hal_enter_critical(void);
```

Usage

NSPE mailbox invokes tfm_ns_mailbox_hal_enter_critical() before entering critical section of NSPE mailbox queue. tfm_ns_mailbox_hal_enter_critical() implementation is platform specific.

tfm_ns_mailbox_hal_enter_critical() should not be called in any interrupt service routine.

tfm_ns_mailbox_hal_exit_critical()

This function exits the critical section of NSPE mailbox queue access.

```
void tfm_ns_mailbox_hal_exit_critical(void);
```

Usage

NSPE mailbox invokes tfm_ns_mailbox_hal_exit_critical() after exiting critical section of NSPE mailbox queue. tfm_ns_mailbox_hal_exit_critical() implementation is platform specific.

tfm_ns_mailbox_hal_exit_critical() should not be called in any interrupt service routine.

tfm_ns_mailbox_hal_enter_critical_isr()

This function enters the critical section of NSPE mailbox queue access in an IRQ handler.

```
void tfm_ns_mailbox_hal_enter_critical(void);
```

Usage

NSPE mailbox invokes `tfm_ns_mailbox_hal_enter_critical_isr()` before entering critical section of NSPE mailbox queue in an IRQ handler. `tfm_ns_mailbox_hal_enter_critical_isr()` implementation is platform specific.

tfm_ns_mailbox_hal_exit_critical_isr()

This function exits the critical section of NSPE mailbox queue access in an IRQ handler

```
void tfm_ns_mailbox_hal_exit_critical_isr(void);
```

Usage

NSPE mailbox invokes `tfm_ns_mailbox_hal_exit_critical_isr()` after exiting critical section of NSPE mailbox queue in an IRQ handler. `tfm_ns_mailbox_hal_exit_critical_isr()` implementation is platform specific.

NSPE mailbox RTOS abstraction APIs

The APIs defined in this section should be implemented by RTOS-specific implementation when `TFM_MULTI_CORE_NS_OS` is enabled.

Note: If `TFM_MULTI_CORE_NS_OS` is set to `OFF`, the following APIs are defined as dummy functions or empty functions.

tfm_ns_mailbox_os_lock_init()

This function initializes the multi-core lock for synchronizing PSA client call(s). The actual implementation depends on the non-secure use scenario.

```
int32_t tfm_ns_mailbox_os_lock_init(void);
```

Return

`MAILBOX_SUCCESS`	Initialization succeeded.
`MAILBOX_GENERIC_ERROR`	Initialization failed.

Usage

`tfm_ns_mailbox_init()` invokes this function to initialize the lock. If `TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD` is enabled, `tfm_ns_mailbox_os_lock_init()` is defined as a dummy one.

tfm_ns_mailbox_os_lock_acquire()

This function acquires the multi-core lock for synchronizing PSA client call(s). The actual implementation depends on the non-secure use scenario.

```
int32_t tfm_ns_mailbox_os_lock_acquire(void);
```

Return

MAILBOX_SUCCESS	Succeeded to acquire the lock.
MAILBOX_GENERIC_ERROR	Failed to acquire the lock.

Usage

tfm_ns_mailbox_client_call() invokes this function to acquire the lock when TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is disabled If TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is enabled, tfm_ns_mailbox_os_lock_acquire() is defined as a dummy one.

tfm_ns_mailbox_os_lock_release()

This function releases the multi-core lock for synchronizing PSA client call(s). The actual implementation depends on the non-secure use scenario.

```
int32_t tfm_ns_mailbox_os_lock_release(void);
```

Return

MAILBOX_SUCCESS	Succeeded to release the lock.
MAILBOX_GENERIC_ERROR	Failed to release the lock.

Usage

tfm_ns_mailbox_client_call() invokes this function to release the lock when TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is disabled If TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is enabled, tfm_ns_mailbox_os_lock_release() is defined as a dummy one.

tfm_ns_mailbox_os_get_task_handle()

This function gets the handle of the current non-secure task executing mailbox functionalities.

```
void *tfm_ns_mailbox_os_get_task_handle(void);
```

Return

Task handle	The non-secure task handle waiting for PSA Client result.

tfm_ns_mailbox_os_wait_reply()

This function performs use scenario and NS OS specific waiting mechanism to wait for the reply of the specified mailbox message to be returned from SPE.

```
void tfm_ns_mailbox_os_wait_reply(void);
```

Usage

The PSA Client API implementations call `tfm_ns_mailbox_os_wait_reply()` to fall into sleep to wait for PSA Client result.

tfm_ns_mailbox_os_wake_task_isr()

This function wakes up the dedicated task which is waiting for PSA Client result, via RTOS-specific wake-up mechanism.

```
void tfm_ns_mailbox_hal_wait_reply(const void *task_handle);
```

Parameters

`task_handle`	The handle to the task to be woken up.

tfm_ns_mailbox_os_mq_create()

This function creates and initializes a NS OS message queue.

```
void *tfm_ns_mailbox_os_mq_create(ize_t msg_size, uint8_t msg_count);
```

Parameters

`msg_size`	The maximum message size in bytes.
`msg_count`	The maximum number of messages in queue.

Return

message queue handle	The handle of the message queue created, or NULL in case of error.

Usage

If `TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD` is disabled, `tfm_ns_mailbox_os_mq_create()` is defined as a dummy one.

tfm_ns_mailbox_os_mq_send()

This function sends PSA Client call request via NS OS message queue.

```
int32_t tfm_ns_mailbox_os_mq_send(void *mq_handle,
                                  const void *msg_ptr);
```

Parameters

mq_handle	The handle of message queue.
msg_ptr	The pointer to the message to be sent.

Return

MAILBOX_SUCCESS	The message is successfully sent.
Other return code	Operation fails with an error code.

Usage

If TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is disabled, tfm_ns_mailbox_os_mq_send() is defined as a dummy one.

tfm_ns_mailbox_os_mq_receive()

This function receives PSA Client call requests via NS OS message queue.

```
int32_t tfm_ns_mailbox_os_mq_receive(void *mq_handle,
                                     void *msg_ptr);
```

Parameters

mq_handle	The handle of message queue.
msg_ptr	The pointer to buffer for message to be received.

Return

MAILBOX_SUCCESS	A message is successfully received.
Other return code	Operation fails with an error code.

Usage

The buffer size must be large enough to contain the request whose size is set in msg_size `` in ``tfm_ns_mailbox_os_mq_create().

If TFM_MULTI_CORE_NS_OS_MAILBOX_THREAD is disabled, tfm_ns_mailbox_os_mq_receive() is defined as a dummy one.

Note: The function caller should be blocked until a PSA Client call request is received from message queue, unless a fatal error occurs.

SPE mailbox APIs

SPE mailbox interface APIs

The APIs defined in this section are called in TF-M routines and platform specific secure interrupt handler.

tfm_mailbox_handle_msg()

This function completes the handling of mailbox messages from NSPE.

```
int32_t tfm_mailbox_handle_msg(void);
```

Return

MAILBOX_SUCCESS	The operation completes successfully.
Other return codes	Operation fails with an error code.

Usage

tfm_mailbox_handle_msg() is registered to RPC callback function handle_req.

tfm_mailbox_handle_msg() executes the following tasks:

- Check NSPE mailbox queue status.
- Copy mailbox message(s) from NSPE. Optional.
- Checks and validations if necessary
- Parse mailbox message
- Call TF-M RPC APIs to pass PSA Client request to TF-M SPM.

tfm_mailbox_reply_msg()

This function replies the PSA Client result to NSPE.

```
int32_t tfm_mailbox_reply_msg(mailbox_msg_handle_t handle, int32_t reply);
```

Parameters

handle	The handle to mailbox message related to the PSA Client result.
reply	The PSA Client result value to be replied.

Return

MAILBOX_SUCCESS	The operation completes successfully.
Other return codes	Operation fails with an error code.

Usage

tfm_mailbox_reply_msg() is registered to RPC callback reply. It is invoked inside handler of psa_reply() to return the PSA Client result to NSPE.

handle determines which mailbox message in SPE mailbox queue contains the PSA Client call. If handle is set as MAILBOX_MSG_NULL_HANDLE, the return result is replied to the mailbox message in the first SPE mailbox queue slot.

tfm_mailbox_init()

This function initializes SPE mailbox.

```
int32_t tfm_mailbox_init(void);
```

Return

MAILBOX_SUCCESS	Initialization succeeds.
Other return codes	Initialization failed with an error code.

Usage

tfm_mailbox_init() invokes tfm_mailbox_hal_init() to execute platform specific initialization.

SPE mailbox HAL APIs

tfm_mailbox_hal_notify_peer()

This function invokes platform specific Inter-Processor Communication drivers to send notification to NSPE.

```
int32_t tfm_mailbox_hal_notify_peer(void);
```

Return

MAILBOX_SUCCESS	The operation completes successfully.
Other return codes	Operation fails with an error code.

Usage

tfm_mailbox_hal_notify_peer() should be implemented by platform specific Inter-Processor Communication drivers.

tfm_mailbox_hal_notify_peer() should not be exported outside SPE mailbox.

tfm_mailbox_hal_init()

This function is implemented by platform support in TF-M. It completes platform specific mailbox initialization, including receiving the the address of NSPE mailbox queue and Inter-Processor Communication initialization.

```
int32_t tfm_mailbox_hal_init(struct secure_mailbox_queue_t *s_queue);
```

Parameters

s_queue	The base address of SPE mailbox queue.

Return

`MAILBOX_SUCCESS`	Initialization succeeds.
Other return codes	Initialization failed with an error code.

`tfm_mailbox_hal_enter_critical()`

This function enters the critical section of NSPE mailbox queue access in SPE.

```
void tfm_mailbox_hal_enter_critical(void);
```

Usage

SPE mailbox invokes `tfm_mailbox_hal_enter_critical()` before entering critical section of NSPE mailbox queue. `tfm_mailbox_hal_enter_critical()` implementation is platform specific.

`tfm_mailbox_hal_enter_critical()` can be called in an interrupt service routine.

`tfm_mailbox_hal_exit_critical()`

This function exits from the critical section of NSPE mailbox queue access in SPE.

```
void tfm_mailbox_hal_exit_critical(void);
```

Usage

SPE mailbox invokes `tfm_mailbox_hal_exit_critical()` when exiting from critical section of NSPE mailbox queue. `tfm_mailbox_hal_exit_critical()` implementation is platform specific.

`tfm_mailbox_hal_exit_critical()` can be called in an interrupt service routine.

Reference

11.2.4 Mailbox NS Agent Design Update

Organization
Arm Limited

Contact
tf-m@lists.trustedfirmware.org

Background

The SPE component that maintains the non-secure clients' request is called 'NS Agent' in TF-M. Besides the Trustzone-based isolation mechanism, there is one other isolation mechanism that implements individual PEs in physically isolated cores respectively. NSPE and SPE transfer non-secure client requests via inter-processor communication based on mailboxes. The component that handles inter-processor communication messages is called `Mailbox NS Agent`.

Note: There may be hardware components and software solutions containing 'mailbox' in their names. The concept `mailbox` in this document represent the mechanism described above, which is not referring to the external concepts.

When the first version `Mailbox NS Agent` was introduced, the generic FF-M interrupt handling was not ready. Hence a customized solution `Multiple Core` is implemented. This customized implementation:

- Perform customized operations on SPM internal data in a deferred interrupt handler.
- Process mailbox operations as special cases in SPM common logic.

These behaviours couple SPM tightly with mailbox logic, which bring issues for maintenance. To address the issue, an updated design shall:

- Make SPM manage other components in a unified way (For example, it is simpler for SPM if all non-SPM components under the IPC model act as `processes`).
- Can use FF-M compliant interrupt mechanism and APIs.

Following the above guidelines makes the `Mailbox NS Agent` work like a `partition`. The agent has an endless loop and waits for signals, calls FF-M API based on the parsing result on the communication messages. But there are still issues after looking closer to the requirements of the agent:

- SPM treats FF-M Client API caller's ID as the client ID. While the mailbox NS agent may represent multiple non-secure clients. Hence it needs to tell SPM which non-secure client it is representing, and the default FF-M Client API does not have such capability.
- FF-M Client API blocks caller before the call is replied; while the mailbox NS Agent needs to respond to the non-secure interrupts in time. Blocking while waiting for a reply may cause the non-secure communication message not to be handled in time.

Extra design items need to be added to address the issues.

Design Update

The below figure shows the overall design to cover various component types. NS Agents are the implementation-defined components that provide FF-M compliant Client API to the non-secure clients. Hence from the view of the non-secure clients, the FF-M client API behaviour follows the FF-M definition. And NS Agent needs customization in SPM since it has extra requirements compared to a generic secure partition.

Figure 4:: Component types and the callable API set

Note: 3 non-SPM component types here: FF-M-compliant Secure Partition (aka `partition`), Trustzone-based NS Agent (aka `Trustzone NS Agent`) and mailbox-based NS Agent (aka `Mailbox NS Agent`). `Trustzone NS Agent` is mentioned here for the comparison purpose. The implementation details for this NS agent type is not introduced here.

To make the programming model close to the FF-M compliance, the `Mailbox NS Agent` is designed as:

- Working like a standard Secure Partition under the IPC model, has one single thread, can call FF-M standard API.
- Having a manifest file to describe the attributes and resources and a positive value `Partition ID` in the manifest.

Services rely on the `client_id` to apply policy-checking, hence SPM needs to know which `client_id` the mailbox NS Agent is representing when mailbox API is calling Client API. The standard API treats the caller as the client of the service, which means that a specific API is required to support identifying the represented non-secure client. SPM sets the non-secure `client_id` into the message right at the moment the message is going to be sent. Before this point, SPM performs the call based on the agent's ID.

This specifc `Agent API` is non-blocking, unlike the standard FF-M Client APIs. This can improve the communication efficiency between NS clients and mailbox NS agents. With this mechanism, extra signals and APIs for message acknowledges are also required.

Note: A standard Secure Partition gets errors when calling the `Agent API`.

Updated programming interfaces

These Client APIs are expanded from the standard Client APIs:

- `agent_psa_connect()` is extended from `psa_connect()`.
- `agent_psa_close()` is extended from `psa_close()`.
- `agent_psa_call()` is extended from `psa_call()`.

And to cooperate with the changed behaviour of these APIs, extra defined signals and types are also involved.

Note: Namespace `agent` is involved for NS Agent callable API; namespace `tfm` is involved for TF-M specific concepts. Even though `agent` is TF-M specific at the current stage, it is proposed to be a common concept for general FF-M compliant implementations, hence assigning `agent` for proposed API and data structures.

Input and output vectors

When non-secure clients call `psa_call()`, a mailbox message containing `psa_call()` parameters is delivered into `Mailbox NS Agent` and the agent needs to extract parameters from the message and then call `agent_psa_call()`. Revisit the `psa_call()` prototype to see the parameters:

```
psa_status_t psa_call(psa_handle_t handle,
                      int32_t type,
                      const psa_invec *in_vec,
                      size_t in_len,
                      psa_outvec *out_vec,
                      size_t out_len);
```

Interface `agent_psa_call()` has 4 arguments only, to avoid ABI complexity when more than 4 arguments get involved. Then input and output vectors must be squashed into a new type to achieve this squashing. There are several scenarios to be considered:

- In the shared-memory-based mailbox scheme, the non-secure interface layer puts `psa_invec` and `psa_outvec` instances and their members in the shared memory. The pointers of these items are delivered to `Mailbox NS Agent` for the agent's referencing. All vectors are non-secure in this case.

- Still, in the shared-memory-based mailbox scheme, but this time not all parameters are prepared by non-secure clients. Some of the items are allocated by the agent. Secure and non-secure vectors get mixed in this case.
- In the scheme that a memory address can not be delivered (A serial interface e.g.), `Mailbox NS Agent` allocates `psa_invec` and `psa_outvec` in local memory to collect pointers and sizes of received buffers. All the vectors are secure in this case.

Based on these scenarios, and the case that an agent might access depending services with the agent itself's identifier and secure vectors, a straight conclusion is that the memory source information - secure or non-secure - for input and output vectors and their pointing memories need to be indicated by the agent so that SPM can perform a memory check towards given vectors.

Most of the SPE platforms have the capability to identify if a given memory pointer is secure or not, which makes this indication look duplicated. But the indication is necessary for these scenarios:

- The SPE platform identifies the memory source by the address mapped into SPE's address space, but this mapping happens inside the agent instead of SPM. It is the agent who receives the mailbox data and maps it, so it knows which addresses need mapping and the others do not.
- The SPE platform just does not have such capability. The addresses from the mailbox can be treated as non-secure always but there are cases that the agent itself needs to access services with its own memory instead of representing the non-secure clients.

To cover the above mentioned scenarios, guidelines are listed for input and output vector processing:

- The agent needs to tell SPM where vectors and descriptors come from, to assist SPM performs a proper memory checking. The source information is encoded in the parameter `control`.
- When SPE platforms have the capability to identify the memory sources, platforms can decide whether to skip the indication or not, in the HAL.

A composition type is created for packing the vectors:

```
struct client_params_t {
  int32_t         ns_client_id_stateless;
  const psa_invec *p_invecs;
  psa_outvec      *p_outvecs;
};
```

`ns_client_id_stateless` indicates the non-secure client id when the client is accessing a stateless service. This member is ignored if the target service is a connection-based one.

Note: The vectors and non-secure client ID are recorded in the internal handle. Hence it is safe to claim `client_params_t` instance as local variable.

Agent-specific Client API

`agent_psa_connect()` and `agent_psa_close()` are the APIs added to support agent forwarding NS requests.

```
psa_handle_t agent_psa_connect(uint32_t sid, uint32_t version,
                               int32_t ns_client_id, const void *client_data);
psa_status_t agent_psa_close(psa_handle_t handle, int32_t ns_client_id);
```

One extra parameter `ns_client_id` added to tell SPM which NS client the agent is representing when API gets called. It is recorded in the handle association data in SPM and requires to be a negative value; ZERO or positive values are invalid non-secure client IDs, SPM does not use these invalid IDs in the message. Instead, it puts the agent's ID into the messaging in this case. This mechanism can provide chances for the agents calling APIs for their own service accessing and API works asynchronously.

As mentioned, the standard FF-M Client service accessing API are blocked until the IPC message gets replied to. While this API returns immediately without waiting for acknowledgement. Unless an error occurred, these agent-specific API returns PSA_SUCCESS always. The replies for these access requests are always fetched initiative by the agent with a `psa_get()`.

```
psa_status_t agent_psa_call(psa_handle_t handle, uint32_t control,
                            const struct client_param_t   *params,
                            const void   *client_data_stateless);
```

Compared to the standard `psa_call()`, this API:

- Squashes the `psa_invec`, `psa_outvec`, and `ns_client_id_stateless` into parameter `params`.
- One extra parameter `client_data_stateless` for `agent_psa_call()` stands for the auxiliary data added. This member is ignored for connection-based services because `agent_psa_connect()` already assigned one in the connected handle.
- Has a composite argument `control`.

The encoding scheme for `control`:

Bit(s)	Name	Description
27	NSIV	*1*: Input vectors in non-secure memory. *0*: Secure memory.
24-26	IVNUM	Number of input vectors.
19	NSOV	*1*: Output vectors in non-secure memory. *0*: Secure memory.
16-18	OVNUM	Number of output vectors.
0-15	type	signed 16-bit service *type*.

Note: `control` is a 32-bit unsigned integer and bits not mentioned in the table are reserved for future usage.

Agent-specific signal

To cooperate with the agent-specific API, one extra acknowledgement signal is defined:

```
#define ASYNC_MSG_REPLY            (0x00000004u)
```

This signal can be sent to agent type component only. An agent can call `psa_get()` with this signal to get one acknowledged message. This signal is cleared when all queued messages for the agent have been retrieved using `psa_get()`. SPM assembles the information into agent provided message object. For the stateless handle, the internal handle object is freed after this `psa_get()` call. The agent can know what kind of message is acknowledged by the `type` member in the `psa_msg_t`, and the `client_data` passed in is put in member `rhandle`. If no 'ASYNC_MSG_REPLY' signals pending, calling `psa_get()` gets `panic`.

Code Example

```
/*
 * The actual implementation can change this __customized_t freely, or
 * discard this type and apply some in-house mechanism - the example
 * here is to introduce how an agent works only.
 */
struct __customized_t {
    int32_t      type;
    int32_t      client_id;
    psa_handle_t handle;
    psa_handle_t status;
};

void mailbox_main(void)
{
    psa_signal_t   signals;
    psa_status_t   status;
    psa_msg_t      msg;
    struct client_param_t client_param;
    struct __customized_t ns_msg;

    while (1) {
        signals = psa_wait(ALL, BLOCK);

        if (signals & MAILBOX_INTERRUPT_SIGNAL) {
            /* NS memory check needs to be performed. */
            __customized_platform_get_mail(&ns_msg);

            /*
             * MACRO 'SID', 'VER', 'NSID', 'INVEC_LEN', 'OUTVEC_LEN', and
             * 'VECTORS' represent necessary information extraction from
             * 'ns_msg', put MACRO names here and leave the details to the
             * implementation.
             */
            if (ns_msg.type == PSA_IPC_CONNECT) {
                status = agent_psa_connect(SID(ns_msg), VER(ns_msg),
                                           NSID(ns_msg), &ns_msg);
            } else if (ns_msg.type == PSA_IPC_CLOSE) {
                status = agent_psa_close(ns_msg.handle, NSID(ns_msg));
            } else {
                /* Other types as call type and let API check errors. */
                client_param.ns_client_id_stateless = NSID(ns_msg);

                /*
                 * Use MACRO to demonstrate two cases: local vector
                 * descriptor and direct descriptor forwarding.
                 */

                /* Point to vector pointers in ns_msg. */
                PACK_VECTOR_POINTERS(client_param, ns_msg);
                status = agent_psa_call(ns_msg.handle,
```

(continues on next page)

(continued from previous page)

```
                                        PARAM_PACK(ns_msg.type,
                                                   INVEC_LEN(ns_msg),
                                                   OUTVEC_LEN(ns_msg))|
                                                   NSIV | NSOV,
                                        &client_param,
                                        &ns_msg);
            }
            /*
             * The service access reply is always fetched by a later
             * `psa_get` hence here only errors need to be dispatched.
             */
            error_dispatch(status);

        } else if (signals & ASYNC_MSG_REPLY) {
            /* The handle is freed for stateless service after 'psa_get'. */
            status        = psa_get(ASYNC_MSG_REPLY, &msg);
            ms_msg        = msg.rhandle;
            ns_msg.status = status;
            __customized_platform__send_mail(&ns_msg);
        }
    }
}
```

Note: `__customized*` API are implementation-specific APIs to be implemented by the mailbox Agent developer.

Customized manifest attribute

Three extra customized manifest attributes are added:

Name	Description
ns_agent	Indicate if manifest owner is an Agent.
client_id_base	The minimum client ID value (<0)
client_id_limit	The maximum client ID value (<0)

`client_id_base` and `client_id_limit` are negative numbers. This means that `client_id_base <= client_id_limit`, but `abs(client_id_base) >= abs(client_id_limit)`. SPM can detect ID overlap when initialising secure partitions

The Non-secure callers are expected to provide a negative (<0) client ID when calling PSA API. A uniform mapping is implemented across all the NS agents, where the mapping is defined as the following:

NS client ID	Transformed client ID
-1	client_id_limit
-2	client_id_limit - 1
…	
-(abs(client_id_limit)-abs(client_id_base)+1)	client_id_base

Any other IDs provided by the NSPE will result in PSA_ERROR_INVALID_ARGUMENT.

Manifest tooling update

The manifest for agents involves specific keys (‘ns_agent’ e.g.), these keys give hints about how to achieve out-of-FF-M partitions which might be abused easily by developers, for example, claim partitions as agents. Some restrictions need to be applied in the manifest tool to limit the general secure service development referencing these keys.

Note: The limitations can mitigate the abuse but can’t prevent it, as developers own all the source code they are working with.

One mechanism: adding a confirmation in the partition list file.

```
"description": "Non-Secure Mailbox Agent",
"manifest": "${CMAKE_SOURCE_DIR}/secure_fw/partitions/ns_agent_mailbox/ns_agent_mailbox.
↪yaml",
"non_ffm_attributes": "ns_agent", "other_option",
```

`non_ffm_attributes` tells the manifest tool that `ns_agent` is valid in ns_agent_mailbox.yaml. Otherwise, the manifest tool reports an error when a non-agent service abuses ns_agent in its manifest.

Runtime programming characteristics

Mailbox agent shall not be blocked by Agent-specific APIs. It can be blocked when:

- It is calling standard PSA Client APIs.
- It is calling `psa_wait()`.

IDLE processing

Only ONE place is recommended to enter IDLE. The place is decided based on the system topologies:

- If there is one Trustzone-based NSPE, this NSPE is the recommended place no matter how many mailbox agents there are in the system.
- If there are only mailbox-based NSPEs, entering IDLE can happen in one of the mailbox agents.

The solution is:

- An IDLE entering API is provided in SPRTL.
- A partition without specific flag can’t call this API.
- The manifest tooling counts the partitions with this specific flag, and assert errors when multiple instances are found.

11.2.5 Memory Access Check of Trusted Firmware-M in Multi-Core Topology

Author
David Hu

Organization
Arm Limited

Contact
david.hu@arm.com

Introduction

TF-M memory access check function `tfm_has_access_to_region()` checks whether an access has proper permission to read or write the target memory region.

On single Armv8-M core platform based on Trustzone, TF-M memory access check implementation relies on Armv8-M Security Extension (CMSE) intrinsic `cmse_check_address_range()`. The secure core may not implement CMSE on multi-core platforms. Even if CMSE is implemented on a multi-core platform, additional check on system-level security and memory access management units are still necessary since CMSE intrinsics and TT instructions are only aware of MPU/SAU/IDAU inside the secure core.

As a result, TF-M in multi-core topology requires a dedicated access check process which can work without CMSE support. This document discuss about the design of the memory access check in multi-core topology.

Overall Design

Memory Access Check Policy

The policies vary in diverse Isolation Levels.

When TF-M works in Isolation Level 1, the access check in multi-core topology checks

- Memory region is valid according to system settings
- Non-secure client call request should not access secure memory.
- Secure services should not directly access non-secure memory. According to PSA Firmware Framework, Secure services should call Secure Partition APIs to ask TF-M SPM to fetch non-secure input/output buffer for them.
- Whether read/write attribute match between access and memory region

When TF-M works in Isolation Level 2, the access check in multi-core topology checks:

- Memory region is valid according to system settings
- Non-secure client call request should not access secure memory.
- Secure services should not directly access non-secure memory. According to PSA Firmware Framework, Secure services should call Secure Partition APIs to ask TF-M SPM to fetch non-secure input/outputs buffer for them.
- Whether read/write attribute match between access and memory region
- Unprivileged secure access should not access privileged secure memory region

The check policy in Isolation Level 3 will be defined according to TF-M future implementation.

The above policies will be adjusted according to TF-M implementation and PSA specs.

General Check Process in TF-M Core

In multi-core topology, tfm_has_access_to_region() is still invoked to keep an uniform interface to TF-M SPM. The function implementation should be placed in multi-core topology specific files separated from Trustzone-based access check.

Multi-core platform specific tfm_hal_memory_check() can invoke tfm_has_access_to_region() to implement the entire memory access check routine.

During the check process, tfm_has_access_to_region() compares the access permission with memory region attributes and determines whether the access is allowed to access the region according to policy described in *Memory Access Check Policy* above.

tfm_has_access_to_region() invokes 3 HAL APIs to retrieve attributes of target memory region.

- tfm_hal_get_mem_security_attr() retrieves the security attributes of the target memory region.
- tfm_hal_get_secure_access_attr() retrieves secure access attributes of the target memory region.
- tfm_hal_get_ns_access_attr() retrieves non-secure access attributes of the target memory region.

All three functions are implemented by multi-core platform support. The definitions are specified in the section *HAL APIs* below.

The pseudo code of tfm_has_access_to_region() is shown below.

```
int32_t tfm_has_access_to_region(const void *p, size_t s, uint8_t flags)
{
    struct security_attr_info_t security_attr;
    struct mem_attr_info_t mem_attr;

    /* Validate input parameters */
    if (Validation failed) {
        return error;
    }

    /* The memory access check should be executed inside TF-M PSA RoT */
    if (Not in privileged level) {
        abort;
    }

    /* Set initial value */
    security_attr_init(&security_attr);

    /* Retrieve security attributes of memory region */
    tfm_hal_get_mem_security_attr(p, s, &security_attr);

    /* Compare according to current Isolation Level */
    if (Input flags mismatch security attributes) {
        return error;
    }

    /* Set initial value */
    mem_attr_init(&mem_attr);

    if (The target memory region is in secure memory space) {
        /* Retrieve access attributes of secure memory region */
```

(continues on next page)

(continued from previous page)

```
        tfm_hal_get_secure_access_attr(p, s, &mem_attr);

        if (Not in Isolation Level 1) {
            /* Secure memory protection unit(s) must be enabled in Isolation Level 2 and 3 */
            if (Protection unit not enabled) {
                abort;
            }
        }
    } else {
        /* Retrieve access attributes of non-secure memory region. */
        tfm_hal_get_ns_access_attr(p, s, &mem_attr);
    }

    /* Compare according to current Isolation Level and non-secure/secure access. */
    if (Input flags match memory attributes) {
        return success;
    }

    return error;
}
```

Note: It cannot be guaranteed that TF-M provides a comprehensive memory access check on non-secure memory for NSPE client call. If non-secure memory protection or isolation is required in a multi-core system, NSPE software should implement and execute the check functionalities in NSPE, rather than relying on TF-M access check.

For example, all the access from NSPE client calls to non-secure memory are classified as unprivileged in current TF-M implementation. Multi-core access check may skip the privileged/unprivileged permission check for non-secure access.

If a multi-core system enforces the privileged/unprivileged isolation and protection of non-secure area, NSPE software should execute the corresponding check functionalities before submitting the NSPE client call request to SPE.

Data Types and APIs

Data Types

Access Permission Flags

The following flags are defined to indicate the access permission attributes. Each flag is mapped to the corresponding CMSE macro. Please refer to ARMv8-M Security Extensions: Requirements on Development Tools for details of each CMSE macro.

MEM_CHECK_MPU_READWRITE

Mapped to CMSE macro CMSE_MPU_READWRITE to indicate that the access requires both read and write permission to the target memory region.

```
#define MEM_CHECK_MPU_READWRITE   (1 << 0x0)
```

MEM_CHECK_MPU_UNPRIV

Mapped to CMSE macro CMSE_MPU_UNPRIV to indicate that it is an unprivileged access.

```
#define MEM_CHECK_MPU_UNPRIV      (1 << 0x2)
```

MEM_CHECK_MPU_READ

Mapped to CMSE macro CMSE_MPU_READ. It indicates that it is a read-only access to target memory region.

```
#define MEM_CHECK_MPU_READ        (1 << 0x3)
```

MEM_CHECK_NONSECURE

Mapped to CSME macro CMSE_NONSECURE to indicate that it is a access from non-secure client call request. If this flag is unset, it indicates the access is required from SPE.

```
#define MEM_CHECK_AU_NONSECURE    (1 << 0x1)
#define MEM_CHECK_MPU_NONSECURE   (1 << 0x4)
#define MEM_CHECK_NONSECURE       (MEM_CHECK_AU_NONSECURE | \
                                   MEM_CHECK_MPU_NONSECURE)
```

Security Attributes Information

The structure security_attr_info_t contains the security attributes information of the target memory region. tfm_hal_get_mem_security_attr() implementation should fill the structure fields according to the platform specific secure isolation setting.

```
struct security_attr_info_t {
    bool is_valid;
    bool is_secure;
};
```

is_valid indicates whether the target memory region is valid according to platform resource assignment and security isolation configurations.

is_secure indicates the target memory region is secure or non-secure. The value is only valid when is_valid is true.

Memory Attributes Information

The structure mem_attr_info_t contains the memory access attributes information of the target memory region. tfm_hal_get_secure_access_attr() and tfm_hal_get_ns_access_attr() implementations should fill the structure fields according to the memory protection settings.

```
struct mem_attr_info_t {
    bool is_mpu_enabled;
    bool is_valid;
    bool is_xn;
    bool is_priv_rd_allow;
    bool is_priv_wr_allow;
    bool is_unpriv_rd_allow;
    bool is_unpriv_wr_allow;
};
```

is_mpu_enabled indicates whether the MPU and other management unit are enabled and work normally.

is_valid indicates whether the target memory region is valid according to platform resource assignment and memory protection configurations.

is_xn indicates whether the target memory region is Execute Never. This field is only valid when is_valid is true.

is_priv_rd_allow and is_priv_wr_allow indicates whether the target memory region allows privileged read/write. Both the fields are valid only when is_valid is true.

is_unpriv_rd_allow and is_unpriv_wr_allow indicates whether the target memory region allows unprivileged read/write. Both the fields are valid only when is_valid is true.

HAL APIs

tfm_hal_get_mem_security_attr()

tfm_hal_get_mem_security_attr() retrieves the current active security configuration information and fills the security_attr_info_t.

```
void tfm_hal_get_mem_security_attr(const void *p, size_t s,
                                   struct security_attr_info_t *p_attr);
```

Parameters	
p	Base address of the target memory region
s	Size of the target memory region
p_attr	Pointer to the security_attr_info_t to be filled
Return	
void	None

The implementation should be decoupled from TF-M current isolation level or access check policy.

All the fields in security_attr_info_t shall be explicitly set in tfm_hal_get_mem_security_attr().

If the target memory region crosses boundaries of different security regions or levels in security isolation configuration, tfm_hal_get_mem_security_attr() should determine whether the memory region violates current security isolation. It is recommended to mark the target memory region as invalid in such case, even if the adjoining regions or levels have the same security configuration.

If the target memory region is not explicitly specified in memory security configuration, `tfm_hal_get_mem_security_attr()` can return the following values according to actual use case:

- Either set `is_valid = false`
- Or set `is_valid = true` and set `is_secure` according to platform specific policy.

`tfm_hal_get_secure_access_attr()`

`tfm_hal_get_secure_access_attr()` retrieves the secure memory protection configuration information and fills the `mem_attr_info_t`.

```
void tfm_hal_get_secure_access_attr(const void *p, size_t s,
                                    struct mem_attr_info_t *p_attr);
```

Parameters	
`p`	Base address of the target memory region
`s`	Size of the target memory region
`p_attr`	Pointer to the `mem_attr_info_t` to be filled
Return	
`void`	None

The implementation should be decoupled from TF-M current isolation level or access check policy.

All the fields in `mem_attr_info_t` shall be explicitly set in `tfm_hal_get_secure_access_attr()`, according to current active memory protection configuration. It is recommended to retrieve the attributes from secure MPU and other hardware memory protection unit(s). The implementation can also be simplified by checking static system-level memory layout.

If the target memory region is not specified in current active secure memory protection configuration, `tfm_hal_get_secure_access_attr()` can select the following values according to actual use case.

- Either directly set `is_valid` to `false`
- Or set `is_valid` to `true` and set other fields according to other memory assignment information, such as static system-level memory layout.

If secure memory protection unit(s) is *disabled* and the target memory region is a valid area according to platform resource assignment, `tfm_hal_get_secure_access_attr()` must set `is_mpu_enabled` to `false` and set other fields according to current system-level memory layout.

`tfm_hal_get_ns_access_attr()`

`tfm_hal_get_ns_access_attr()` retrieves the non-secure memory protection configuration information and fills the `mem_attr_info_t`.

```
void tfm_hal_get_ns_access_attr(const void *p, size_t s,
                                struct mem_attr_info_t *p_attr);
```

Parameters	
p	Base address of the target memory region
s	Size of the target memory region
p_attr	Pointer to the mem_attr_info_t to be filled
Return	
void	None

The implementation should be decoupled from TF-M current isolation level or access check policy.

Since non-secure core runs asynchronously, the non-secure MPU setting may be modified by NSPE OS and therefore the attributes of the target memory region can be unavailable during `tfm_hal_get_ns_access_attr()` execution in TF-M. When the target memory region is not specified in non-secure MPU, `tfm_hal_get_ns_access_attr()` can set the fields according to other memory setting information, such as static system-level memory layout.

If non-secure memory protection unit(s) is *disabled* and the target memory region is a valid area according to platform resource assignment, `tfm_hal_get_ns_access_attr()` can set the following fields in `mem_attr_info_t` to default values:

- `is_mpu_enabled = false`
- `is_valid = true`
- `is_xn = true`
- `is_priv_rd_allow = true`
- `is_unpriv_rd_allow = true`

`is_priv_wr_allow` and `is_unpriv_wr_allow` can be set according to current system-level memory layout, such as whether it is in code section or data section.

General retrieval functions

TF-M implements 3 general retrieval functions to retrieve memory region security attributes or memory protection configurations, based on static system-level memory layout. Platform specific HAL functions can invoke those 3 general functions to simplify implementations.

- `tfm_get_mem_region_security_attr()` retrieves general security attributes from static system-level memory layout.
- `tfm_get_secure_mem_region_attr()` retrieves general secure memory protection configurations from static system-level memory layout.
- `tfm_get_ns_mem_region_attr()` retrieves general non-secure memory protection configurations from static system-level memory layout.

If a multi-core platform's memory layout may vary in runtime, it shall not rely on these 3 functions to retrieve static configurations. These 3 functions run through memory layout table to check against each memory section one by one, with pure software implementation. It might cost more time compared to hardware-based memory access check.

tfm_get_mem_region_security_attr()

tfm_get_mem_region_security_attr() retrieves security attributes of target memory region according to the static system-level memory layout and fills the security_attr_info_t.

```
void tfm_get_mem_region_security_attr(const void *p, size_t s,
                                      struct security_attr_info_t *p_attr);
```

Parameters	
p	Base address of the target memory region
s	Size of the target memory region
p_attr	Pointer to the security_attr_info_t to be filled
Return	
void	None

tfm_get_secure_mem_region_attr()

tfm_get_secure_mem_region_attr() retrieves general secure memory protection configuration information of the target memory region according to the static system-level memory layout and fills the mem_attr_info_t.

```
void tfm_get_secure_mem_region_attr(const void *p, size_t s,
                                    struct mem_attr_info_t *p_attr);
```

Parameters	
p	Base address of the target memory region
s	Size of the target memory region
p_attr	Pointer to the mem_attr_info_t to be filled
Return	
void	None

tfm_get_ns_mem_region_attr()

tfm_get_ns_mem_region_attr() retrieves general non-secure memory protection configuration information of the target memory region according to the static system-level memory layout and fills the mem_attr_info_t.

```
void tfm_get_ns_mem_region_attr(const void *p, size_t s,
                                struct mem_attr_info_t *p_attr);
```

Parameters	
p	Base address of the target memory region
s	Size of the target memory region
p_attr	Pointer to the mem_attr_info_t to be filled
Return	
void	None

11.3 Secure Services

11.3.1 Secure Partition Manager

This document describes the Secure Partition Manager (*SPM*) implementation design in Trusted Firmware-M (*TF-M*).

Note:

- The FF-M in this document refers to the accumulated result of two specifications: FF-M v1.1 Update on FF-M v1.0.
- The words marked as *interpreted* are defined terms. Find the terms in referenced documents if it is not described in this document.

Introduction

The service access process of FF-M:

Figure 5:: FF-M service access process

Secure services (aka *Service*) is the component providing secure functionalities in *SPE*, and *Client* is the user of the *Service*. A service acts as a client when it is accessing its depending services.

Services are grouped into *Secure Partition* (aka *partition*). A partition:

- Contains services with the same purpose.
- Provides implementation required isolation boundaries.
- Is a software development unit.

Each service exposes its *Service ID* (*SID*) and *Handle* for client access usage. Clients access services by *SID* or *Handle* through FF-M *Client API*. Partitions use FF-M *Secure Partition API* when it needs to operate on client data or reply to a client.

SPM is the centre of an FF-M compliant implementation, which sets up and maintains a firmware framework that:

- Implements *Client API* and *Secure Partition API*.
- Manages partition runtime to follow FF-M.
- Involves necessary implementation-defined items to support the implementation.

SPM interfaces consist of these two categories:

- FF-M defined API.
- Extended API to support the implementation.

Both API categories are compliant with FF-M concepts and guidelines. The core concept of TF-M SPM surrounds the FF-M defined service management and access process. Besides this, another important implementation part is partition runtime management.

Partition runtime model

One partition must work under as *ONE* of the runtime models: *Inter-process communication* (*IPC*) model or *Secure Function* (*SFN*) model.

A partition that runs under the *IPC* model looks like a classic *process*. There is *ONE* thread inside the partition keeps waiting for signals. SPM converts the service accessing info from the *Client API* call into messages and assert a signal to the partition. The partition calls corresponded service function indicated by the signal and its bound message, and reply service returned result to the client. The advantages of this model:

- It provides better isolation by limiting the interfaces on data interactive. Data are preferred to be processed in a local buffer.
- It provides a mechanism for handling multiple service access. There is no memory mapping mechanism in the MCU system, hence it is hard to provide multiple function call contexts when serving multiple-threaded clients if the service access is implemented in a function-call based mechanism. This model converts multiple service accesses into messages, let the partition handles the service access in messages one by one.

The *Secure Function* (*SFN*) model partition is close to a *library*. Each service is provided as a function entry inside the partition. SPM launches the target service function after the service is found. The whole procedure (from client to service function) is a function call. This model:

- Saves the workloads spent on *IPC* scheduling.

Meanwhile, it relaxes the data interactive mechanism, for example, allow direct memory access (MMIOVEC). And it is hard to enable multiple-threaded clients service access because of multiple call context-maintenance difficulties.

An implementation contains only *SFN* partitions fits better in the resource-constrained devices, it is called an *SFN model implementation*. And it is an *IPC model implementation* when *IPC* partitions exist in the system.

Note: *IPC model implementation* can handle access to the services in the *SFN* partition.

Components and isolation levels

There are *THREE* isolation levels defined in *FF-M*. These levels can fulfil different security requirements by defining different isolation boundaries.

Figure 6:: Components and isolation boundaries

Note: Concept *ARoT*, *PRoT*, *domain*, and boundaries are in the *FF-M* specification.

Not like an *SPE* client that can call *Client API* to access the secure services in one step, an *NSPE* client needs to cross the secure boundaries first before calling *Client API*. The component *NS Agent* in Figure 6: represents *NSPE* clients after they crossed the secure boundaries. This could help *SPM* handles the request in a unified way instead of care about the special boundaries.

Note: *NS Agent* is a necessary implementation-defined component out of FF-M specification. *NS Agent* has a dedicated stack because secure and non-secure can not share the stack. It also has dedicated execution bodies. For example, RPC-based *NS Agent* has a while loop that keeps waiting for messages; and Trustzone-based *NS Agent* has veneer code to take over *NSPE* secure call. This makes *NS Agent* to be a component more like a *process*. Hence in the simplest implementation (*SFN model implementation* mentioned above), *NS Agent* is the only process in the system, the scheduling logic can be extremely simplified since no other process execution needs to be scheduled. But the scheduling interface is still necessary to SPM, this could help SPM treat both *SFN* and *IPC* model implementation in a unified way.

Check *NS Agent* for details.

Implementation principle

The principles for TF-M SPM implementation:

Important:

- SPM can treat these components as the client: NS Agent, SFN Partition, and IPC partition.
- These components can provide services: SFN Partition, IPC partition, and built-in services. A built-in service is built up with SPM together.
- All partition services must be accessed by *Client API*.
- Partitions interact with client data by *Secure Partition API*.
- Built-in services are strongly recommended to be accessed by *Client API*. Customized interfaces are restricted.
- Built-in services can call SPM internal interfaces directly.

Runtime management

The runtime execution runs among the components, there are **4** runtime states:

- *Initializing* state, to set up the SPM runtime environment after system powers up
- *IDLE* state, when SPM runtime environment is set up and partitions are ready for service access.
- *Serving* state, when partition is under initializing or service access handling.
- *Background* state, such as the arrival of secure interrupt or unexpected faults. *Background* state returns to the state it preempts. *Background* state can be nested.

The state transition diagram:

Figure 7:: SPE runtime execution states

Initializing

The goal of TF-M initializing is to perform necessary initialization and move to the *Serving* state. This state starts with platform-specific power on sequence, then *SPM* takes over the execution to perform these operations:

1. A preparation initialization process before SPM runtime initialization.
2. SPM runtime initialization.
3. A post initialization happens after the SPM runtime initialization and before the first partition gets launched.

Note: These procedures and their sub-routines are recommended to be applied with execution measurement mechansim to mitigate the *Hardware Fault Injection* attack.

Preparation initialization

The purpose of this preparation initialization is to provide a chance for performing those security required but generic platform power-on skipped operations, such as:

- Restrict *SPM* execution, for example, set up memory overflow settings for SPM runtime memory, or set code out of SPM as un-executable, even though SPM is a privileged component in general.

Note: The `logging`-related peripheral can be set up **AT THIS STEP**, if logging is enabled and it needs peripheral support. There is no standalone initializing HAL API proposed for logging, so here is an ideal place for initializing them.

This procedure is abstracted into one *HAL*, and a few example procedures are implemented as its sub-routines for reference:

- Architecture extensions initialization, Check chapter *Architecture security settings* for detailed information.
- Isolation and lifecycle initialization.

The load isolation boundaries need to be set up here, such as SPE/NSPE boundary, and ARoT/PRoT boundary if isolation level 2 is applied.

The lifecycle is initiated by a secure bootloader usually. And in this stage of SPM initializing, SPM double-checks the lifecycle set up status (following a specific lifecycle management guidelines). Note that the hardware debugger settings can be part of lifecycle settings.

Important: Double-check debugger settings when performing a product release.

SPM runtime initialization

This procedure initializes necessary runtime operations such as memory allocator, loading partitions and partition-specific initialization (binding partitions with platform resources).

The general processes:

1. Initialize runtime functionalities, such as memory allocator.
2. Load partitions by repeating below steps:
    - Find a partition load information.
    - Allocate runtime objects for this partition.
    - Link the runtime objects with load information.
    - Init partition contexts (Thread and call context).
    - Init partition isolation boundaries (MMIO e.g.).
    - Init partition interrupts.

After no more partitions need to be loaded, the SPM runtime is set up but partitions' initialization routines have not run yet - the partition runtime context is initialized for the routine call.

The partition initialization routine is a special service that serves SPM only, because:

- SPM needs to call the initialization routine, just like it calls into the service routine.

- The partition initialization routine can access its depending services. Putting initialization routine in the same runtime environment as common service routines can avoid special operations.

Hence a *Partition initialization client* needs to be created to initialize the SFN partitions, because:

- *SPM runtime initialization* happen inside a special runtime environment compare to the partition runtime execution, then an environment switching is needed.
- IPC partitions are initialized by the scheduler and dependencies are handled by signals and messages asynchronously, hence IPC partitions can process the dependencies by their own.

The *Partition initialization client* is created differently based on the implementation runtime model:

- A SFN client is created under the SFN model implementation.
- A IPC client is created under the IPC model implementation. This client thread has the highest priority.

As the other partitions, the client is created with context standby, and it is executed after the *Post initialization* stage.

Post initialization

Platform code can change specific partition settings in this procedure before partitions start. A few SPM API is callable at this stage, such as set a signal into a specific partition, or customized peripheral settings.

Serving

Two execution categories work under this state:

- *Partition initialization routine execution*.
- *Secure service access*.

This state indicates the serving is ongoing. It is mainly the service routine execution, plus a few SPM executions when SPM API gets called.

Important: The service access process introduced in this chapter (Such as *Secure service access*) is abstracted from the FF-M specification. Reference the FF-M specification for the details of each step.

Partition initialization routine execution

The partition initialization routines get called. One partition may access its depending services during initializing, then this procedure is a *Secure service access*.

The initialization routine gets called initially by *Partition initialization client*, also can be called by Client API before service access, if the target partition is not initialized but a service access request is raised by one client.

Secure service access

The process of service access:

1. A *client* calls an FF-M Client API.
2. *SPM* validates inputs and looks up for the targeted service.
3. *SPM* constructs the request to be delivered under a proper runtime mechanism.
4. The target service gets executed. It can perform internal executions or access depending services to prepare the response. It also can wait for specific signals.
5. The target service calls FF-M Secure Partition API to request a reply to the client.
6. SPM delivers the response to the client, and the API called by the client returns.

The mechanism of how SPM interact with the target partition depends on the partition runtime model.

- Access to a service in an SFN partition is a function call, which does not switch the current process indicator.
- Access to a service in an IPC partition leads to scheduling, which switches the current process indicator.
- When the execution roams between components because of a function call or scheduling, the isolation boundaries NEED to be switched if there are boundaries between components.

No matter what kind of partition a client is trying to access, the SPM API is called firstly as it is the interface for service access. There are two ABI types when calling SPM API: Cross-boundary or No-cross-boundary.

Calling SPM API

SPM is placed in the PRoT domain. It MAY have isolation boundaries under particular isolation levels. For example:

- There are boundaries between ARoT components and SPM under isolation level 2 and 3.

The API SPM provided needs to support the function call (no boundary switching) and cross-boundary call. A direct call reaches the API entrance directly, while a cross-boundary call needs a mechanism (Supervisor call e.g.) to cross the boundary first before reaching the API entrance.

Figure 8:: SPM call types

SPM internal execution flow

SPM internal execution flow as shown in diagram:

Figure 9:: SPM API runtime

The process:

- PSA API gets called by one of the ABI mentioned in the last chapter as *ABI 1* in the diagram.
- The unified API Handler calls FF-M and backend subroutines in sequence.
- The *FF-M* subroutine performs *FF-M* defined operations.
- The backend operations perform target partition runtime model decided operations. For example, enqueue message into the target partition under the IPC runtime model, or prepare to call context with the message as the parameters under the SFN runtime model.

- API Handler triggers different ABI based on the result of the backends.

The API handler:

- Can process the *PROGRAMMER_ERROR* in a unified place.
- Can see the prepared caller and callee context, with exited SPM context. It is an ideal place for subsequent operations such as context switching.

An example code:

```
void abi(void *p)
{
    status = spm_api(p);
    /*
     * Now both the caller and callee contexts are
     * managed by spm_api.
     */
    if (status == ACTION1) {
        /*
         * Check if extra operations are required
         * instead of a direct return.
         */
        exit_action1();
    }
}
```

The explanation about *Scheduler Lock*:

Some FF-M API runs as a generic thread to prevent long time exclusive execution. When a preemption happens, a new partition thread can call SPM API again, makes SPM API nested. It needs extra memory in SPM to be allocated to store the preempted context. Lock the scheduler while SPM API is executing can ensure SPM API complete execution after preemption is handled. There can be multiple ways to lock the scheduler:

- Set a scheduler lock.
- Set SPM API thread priority as the highest.

Backend service messaging

A message to service is created after the target service is found and the target partition runtime model is known. The preparation before ABI triggers the final accessing:

- The message is pushed into partition memory under a specific ABI mechanism if the target partition model is *SFN* and there are boundaries between SPM and the target partition. After this, requests a specific call type to the SPM ABI module.
- The target service routine is called with the message parameter if there are no boundaries between SPM and the target partition and the partition runtime is *SFN*.
- The message is queued into the partition message list if the target partition runtime model is *IPC*.
- IPC partition replies to the client by *psa_reply*, which is another SPM API call procedure.
- SFN partition return triggers an implied *psa_reply*, which is also another SPM API call procedure.

Note: The backends also handle the isolation boundary switching.

Sessions and contexts

FF-M API allows multiple sessions for a service if the service is classic connection-based. The service can maintain multiple local session data and use *rhandle* in the message body to identify which client this session is bound with.

But this does not mean when an ongoing service accessing is preempted, another service access request can get a chance for new access. This is because of the limited context storage - supporting multiple contexts in a common service costs much memory, and runtime operations (allocation and re-location). Limited the context content in the stack only can mitigate the effort, but this requirement requires too much for the service development.

The implementation-decisions are:

- IPC partitions handles messages one by one, the client gets blocked before the service replying to the client.
- The client is blocked when accessing services are handling a service request in an SFN partition.

ABI type summary

The interface type is decided by the runtime model of the target component. Hence PSA API has two types of ABI: *Cross-boundary ABI* and *Function call ABI*. After SPM operations, one more component runtime type shows up: The IPC partition, hence *schedule* is the mechanism when accessing services inside an IPC partition.

Figure 10:: ABI types

Note: The API that does not switch context returns directly, which is not covered in the above diagram.

IDLE state

The *IDLE state* can be represented by the *NS Agent* action:

- Launching NSPE software (Trustzone case, e.g.), or send a signal to NSPE software (RPC case, e.g.).

It is because *NS Agent* is the last component being initialized in the system. Its execution indicates other partitions' initialization has accomplished.

Background state

Background execution can happen at any time when the arrival of interrupts or execution faults. An ongoing background execution indicates the state is a *Background state*. The characteristics:

- The background state has a higher execution priority than other states - other states stall when the background state is executing.
- Background execution can be nested. For example, an interrupt handler can preempt an ongoing interrupt execution.
- Particular partition code can be involved in the background state, for example, the *First Level Interrupt Handler (FLIH)* of one partition.
- Background state MUST return to the state it preempts.

Note: Interrupt handling is a common background state example. Check Interrupt design document for details.

Practical implementation items

This chapter describes the practical implementation contents.

Important: Arm M-profile architecture is the default hardware architecture when describing architecture-specific items.

The general M-profile programming is not involved in this document. The following chapters introduce the mandatory settings for security requirements.

Architecture security settings

When an *Armv8m Security Extension* (Aka *Trustzone-M*) is available in the system, these settings are required to be set:

- The MSPLIM needs to be set correctly to prevent stack overflow.
- The exception handler priority needs to be decided.
- Boost the secure handler mode priority to prevent NSPE from preempting SPE handler mode execution(*AIRCR.PRIS*).
- Disable NSPE hardware faults when a secure fault is happening. Trap in the secure fault with the highest priority can be a valid option.
- Push seals on the stack top when a stack is allocated (*TFMV-1*). Also check *Stack seal* chapter for details.

Besides *Armv8m Security Extension*, these settings need to care when *Floatpoint Extension* is enabled for partition usage:

- *FPCCR.TS*, *FPCCR.CLRONRET* and *FPCCR.CLRONRETS* need to be set when booting.
- *CPACR.CP10* and *CPACR.CP11* need to be set when booting.

Important: Floatpoint usage is prohibited in SPM and background execution.

Stack seal

When Trustzone-M is applied, the architecture specification recommends sealing the secure stack by:

- Push two *SEAL* values (*0xFEF5EDA5*) at the stack bottom, when a stack is allocated.
- Push two *SEAL* values on the stack pointer which is going to be switched out.

Check architecture specification and vulnerability *TFMV-1* for details.

Trustzone-M reentrant

The Trustzone-M has characteristics that:

- SPE keeps the last assigned stack pointer value when execution leaves SPE.
- SPE execution can be preempted by NSPE which causes an execution left.

It is possible that NSPE preemption caused a second thread calls into SPE and re-uses the secure stack contains the first thread's context, which obviously causes information leakage and runtime state inconsistent.

Armv8.1-M provides the hardware setting *CCR_S.TRD* to prevent the reentrant. On an Armv8.0-M architecture, extra software logic needs to be added at the veneer entry:

- Check if the local stack points to a *SEAL* when veneer code get executed.

```
/* This is a theoretical code that is not in a real project. */
veneer() {
    content = get_sp_value();
    if (context != SEAL) /* Error if reentrant detected */
        error();
}
```

SPM Runtime ABI

This chapter describes the runtime implementation of SPM.

Scheduling

The scheduling logic is put inside the PendSV mode. PendSV mode's priority is set as one level higher than the default thread mode priority. If *Trustzone-M* is present, the priority is set as the lowest just above NS exception priority to prevent a preemption in secure exceptions.

PendSV is an ideal place for scheduling logic, because:

- An interrupt triggered scheduling during PendSV execution lead to another PendSV execution before exception return to the thread mode, which can find the latest run-able thread.

Function call ABI

In the diagram Figure 9:, the ABI can have two basic types: cross-boundary and direct call (No-cross-boundary).

When applying *SVCall* (*SVC*) as the cross-boundary mechanism, the implementation can be straight like:

- The SVC handler calls SPM internal routines, and eventually back to the handler before an exit.

Under the IPC model implementation, to re-use *ABI 2* in *No-cross-boundary*, a software ABI needs to be provided.

While under the SFN model plus isolation level 1, both *ABI 1* and *ABI 2* can be a direct function call.

NS Agent

The *NS Agent* (*NSA*) forwards NSPE service access request to SPM. It is a special *partition* that:

- It does not provide FF-M aligned secure services.
- It runs with the second-lowest priority under *IPC model implementation* (The IDLE thread has the lowest priority).
- It has isolation boundaries and an individual stack.
- It requires specific services and mechanisms compared to common partitions.

There are two known types for NS Agent:

- Trustzone-M based.
- Remote Procedure Call (RPC) based.

This process is put inside the ARoT domain, to prevent assign unnecessary PRoT permissions to the NSPE request parsing logic.

Trustzone-M specific

The functionalities of a Trustzone-M specific NSA is:

- Launch NSPE when booting.
- Wait in the veneer code, and get executed when NSPE accesses services.

As there may be multiple NSPE threads calling into SPE, and SPM wants to identify them, special mechanisms can be proposed to provide the identification. Check specific NS ID client ID or context related documents for details.

Figure 11:: TZ NSA and specific service

RPC specific

Compared to Trustzone-M NSA, RPC NSA looks closer to a generic partition:

- It has a message loop, keep waiting for RPC events.
- It converts received RPC events into FF-M API call to target services.

And compared to generic partitions, the differences are:

- It parses RPC messages to know which NSPE thread is accessing services. Hence it needs special interfaces to help SPM to identify the NSPE clients.
- It needs to check NSPE client memory and map to local before calling SPM API.
- It cannot be blocked during API calls, which affects handling the RPC requests.

Partition

A partition is a set of services in the same scope. Services are generally implemented as functions, and the partition exposes the services in different ways based on the partition model: *IPC* or *SFN*.

A partition build generates these outputs:

- A partition load information, used by SPM.
- A partition program containing service interface and logic, typically a library.
- An optional service API set for easier client usage, by encapsulating the low-level *FF-M* Client API. These API needs to be integrated into client space.

Partition loading

SPM needs to set up runtime objects to manage partitions by parsing the load information of partitions. In general, the partition load information is stored in a const memory area can be random read directly, hence SPM can direct link runtime objects to the load information without a copy operation. This is called a *Static Load* mechanism.

Each partition has different numbers of dependencies and services, this makes the load information size of each partition different, it would be hard to put such variable size elements in an array. The solution here is putting these elements in a dedicated section, for SPM enumerating while loading. Each partition can define variable size load information type based on the common load info type.

The common load information:

```
struct partition_load_info_t {
    uint32_t        psa_ff_ver;     /* Encode the version with magic    */
    int32_t         pid;            /* Partition ID                     */
    uint32_t        flags;          /* ARoT/PRoT, SFN/IPC, priority     */
    uintptr_t       entry;          /* Entry point                      */
    size_t          stack_size;     /* Stack size                       */
    size_t          heap_size;      /* Heap size                        */
    uint32_t        ndeps;          /* Dependency number                */
    uint32_t        nservices;      /* Service number                   */
    uint32_t        nassets;        /* Asset numbers                    */
    uint32_t        nirqs;          /* Number of IRQ owned by Partition */
};

And the example for a specific partition load info:
struct partition_example_load_info_t {
    struct partition_load_info_t ldi;      /* Common info info          */
    uint32_t                     deps[10]; /* Dependencies              */
    /*                    ... other infos ...                           */
};
```

Peripheral binding

A partition can declare multiple peripherals (Interrupts are part of peripherals). The peripherals binding process:

- The tooling references symbols in a fixed pattern in the partition load information.
- The HAL implementation needs to provide the symbols being referenced.
- SPM calls HAL API to bind the partition info with devices when the partition gets loaded.
- The platform HAL acknowledges the binding if validation pass on SPM given load information.

Integration and development

These modules are expected to be object/library level modularised, each module should be generated into object/library at build time:

Table 40:: Object level modularization

Name	Description
SPM	All SPM related modules such as SPM, system, and so on.
Platform	Platform sources are switchable.
Services and Secure Partition	These items should be standalone.
Service Runtime Library	This is a shared runtime library.

HAL

The HAL here mainly refers to the SPM HAL. The SPM HAL implementation is running with the same privilege level and hardware mode with SPM. The implementation is object level modularized with SPM.

Check the *HAL* design document for details.

Configurations

The same TF-M code base is flexible to address different implementation requirements, from the simplest device with isolation level 1 to the most complicated device with isolation level 3 and optional isolation rules.

These configurations are set by switches, during the build time, as runtime support costs extra resources. The common configurations are named *profile*. There are several profiles defined.

History

Table 41:: Revision

Date	Description
2021 Apr-Sep	Updated to cover the implementation for *FF-M v1.1* features.
2018	Created as 'TF-M Inter-Process Communication' which is deprecated as this document covers whole SPM content.

11.3.2 Secure Partition Runtime Library

Organization
Arm Limited

Contact
tf-m@lists.trustedfirmware.org

Background

Trusted Firmware - M (TF-M) uses a toolchain provided runtime library and supervisor calls to easily implement the PSA Firmware Framework (PSA FF) API. This working model works well under isolation level 1 since there are no data isolation requirements. While TF-M is evolving, this model is not suitable because:

- The high-level isolation requires isolating data but some toolchain library interfaces have their own global data which cannot be shared between the Secure Partitions.
- The toolchain libraries are designed without taking security as a core design principle.

A TF-M specific runtime library is needed for the following reasons:

- Easier evaluation or certification by security standards.
- Source code transparency.
- Sharing code to save ROM and RAM space for TF-M.

PSA FF specification also describes the requirements of C runtime API for Secure Partitions.

This runtime library is named the `Secure Partition Runtime Library`, and the abbreviation is `SPRTL`.

Design Principle

The following requirements are mandatory for SPRTL implementation:

Important:

- **CODE ONLY** - No read-write data should be introduced into runtime library implementation.
- **Thread safe** - All functions are designed with thread-safe consideration. These APIs access caller stack and caller provided memory only.
- **Isolation** - Runtime API code is set as executable and read-only in higher isolation levels.
- **Security first** - SPRTL is designed for security and it may come with some performance loss.

API Categories

Several known types of functions are included in SPRTL:

- C runtime API.
- RoT Service API.
- PSA Client and Service API.
- [Future expansion, to be detailed later] other secure API.

Security Implementation Requirements

If `malloc/realloc/free` are provided, they must obey additional requirements compared to the C standard: newly allocated memory must be initialized to ZERO, and freed memory must be wiped immediately in case the block contains sensitive data.

The comparison API ('memcmp' e.g.), they should not return immediately when the fault case is detected. The implementation should execute in linear time based on input to avoid execution timing side channel attack.

The pointer validation needs to be considered. In general, at least the 'non-NULL' checking is mandatory. A detection for invalid pointer leads to a `psa_panic()`.

The following section describes the first 3 API types and the implementation requirements.

C Runtime API

PSA FF describes a small set of the C standard library. Part of toolchain library API can be used as default if these APIs meet the *Design Principle* and *Security Implementation Requirements*. The toolchain 'header' and 'types' can be reused to simplify the implementation.

These APIs can take the toolchain provided version, or separately implemented in case there are extra requirements:

Note:

- 'memcpy()/memmove()/memset()'
- String API

These APIs are proposed to be implemented with the security consideration mentioned in *Security Implementation Requirements*:

Note:

- 'memcmp()'
- Other comparison API if referenced ('strcmp' e.g.).

The following functions are optional, but if present, they must conform to additional *Security Implementation Requirements*:

Note:

- 'malloc()/free()/realloc()'
- 'assert()/printf()'

The following APIs are coupled with toolchain library much so applying toolchain library implementation is recommended:

Note:

- Division and modulo - arithmetic operations.
- Other low level or compiler specific functions (such as 'va_list').

Besides the APIs mentioned above, the following runtime APIs are required for runtime APIs with private runtime context (‘malloc’ e.g.):

Note:

- ‘__sprtmain()’ - partition entry runtime wrapper.

RoT Service API

The description of RoT Service API in PSA FF:

Note: Arm recommends that the RoT Service developer also defines an RoT Service API and implementation to encapsulate the use of the IPC protocol, and improve the usability of the service for client firmware.

Part of the RoT Service API have proposed specifications, such as the PSA Cryptography API, PSA Storage API, and PSA Attestation API. It is suggested that the service developer create documents of their RoT Service API and make them publicly available.

The RoT Service API has a large amount and it is the main part of SPRTL. This chapter describes the general implementation of the RoT Service API and the reason for putting them into SPRTL.

In general, a client uses the PSA Client API to access a secure service. For example:

```c
/* Example, not a real implementation */
caller_status_t psa_example_service(void)
{
  ...
  handle = psa_connect(SERVICE_SID, SERVICE_VERSION);
  if (INVALID_HANDLE(handle)) {
    return INVALID_RETURN;
  }

  status = psa_call(handle, type, invecs, inlen, outvecs, outlen);

  psa_close(handle);

  return TO_CALLER_STATUS(status);
}
```

This example encapsulates the PSA Client API, and can be provided as a simpler and more generic API for clients to call. It is not possible to statically link this API to each Secure Partition because of the limited storage space. The ideal solution is to put it inside SPRTL and share it to all Secure Partitions. This would simplify the caller logic into this:

```c
if (psa_example_service() != STATUS_SUCCESS) {
  /* do something */
}
```

This is the simplest case of encapsulating PSA Client API. If a RoT Service API is connect heavy, then, the encapsulation can be changed to include a connection handle inside a context data structure. This context data structure type is defined in RoT Service headers and the instance is allocated by API caller since API implementation does not have private data.

Note:

- Even the RoT Service APIs are provided in SPRTL for all clients, the SPM performs the access check eventually and decides if the access to service can be processed.
- For those RoT Service APIs only get called by a specific client, they can be implemented inside the caller client, instead of putting it into SPRTL.

PSA Client and Service API

Most of the PSA APIs can be called directly with supervisor calls. The only special function is `psa_call`, because it has **6** parameters. This makes the supervisor call handler complex because it has to extract the parameters from the stack. The definition of psa_call is the following:

```
psa_status_t psa_call(psa_handle_t handle, int32_t type,
                      const psa_invec *in_vec, size_t in_len,
                      psa_outvec *out_vec, size_t out_len);
```

The parameters need to be packed to avoid passing parameters on the stack, and the supervisor call needs to unpack the parameters back to **6** for subsequent processing.

Privileged Access Supporting

Due to specified API (printf, e.g.) need to access privileged resources, TF-M Core needs to provide interface for the resources accessing. The permission checking must happen in Core while caller is calling these interface.

Secure Partition Local Storage

There are APIs that need to reference specific partition private data ('malloc' references local heap, e.g.), and the APIs reference the data by mechanisms other than function parameters. The mechanism in TF-M is called 'Secure Partition Local Storage'.

A straight way for accessing the local storage is to put the local storage pointer in a known position in the stack, but there is a bit of difficulty in particular scenarios.

Note:

- The partition's stack is not fixed-size aligned, using stack address aligning method can not work.
- It requires privileged permission to *access* special registers such as *PSPLIMIT*. And Armv6-M and Armv7-M don't have *PSPLIMIT*.`

Another common method is to put the pointer in one shared global variable, and the scheduler maintains the value of this variable to point to the running partition's local storage in runtime. It does not fully align with SPRTL design prerequisites listed above, hence extra settings are required to guarantee the isolation boundaries are not broken.

Important:

- This variable is put inside a dedicated shared region and it can not hold information not belonging to the owner.

And this mechanism has disadvantages:

- It needs extra maintenance effort from the scheduler and extra resources for containing the variable.

TF-M chooses this common way as the default option for local storage and can be expanded to support more methods.

Tooling Support on Partition Entry

PSA FF requires each Secure Partition to have an entry point. For example:

```
/* The entry point function must not return. */
void entry_point(void);
```

Each partition has its own dedicated local_storage for heap tracking and other runtime state. The local_storage is designed to be saved at the read-write data area of a partition with a specific name. A generic entry point needs to be available to get partition local_storage and do initialization before calling into the actual partition entry. This generic entry point is defined as '__sprtmain':

```
void __sprtmain(void)
{
  /* Get current SP private data from local storage */
  struct p_sp_local_storage_t *m =
          (struct p_sp_local_storage_t *)tfm_sprt_local_storage;

  /* Potential heap init - check later chapter */
  if (m->heap_size) {
    m->heap_instance = tfm_sprt_heap_init(m->heap_sa, m->heap_sz);
  }

  /* Call thread entry 'entry_point' */
  m->thread_entry();

  /* Back to tell Core end this thread */
  SVC(THREAD_EXIT);
}
```

Since SPM is not aware of the '__sprtmain' in SPRTL, it just calls into the entry point listed in partition runtime data structure. And the partition writer may be not aware of running of '__sprtmain' as the generic wrapper entry, tooling support needs to happen to support this magic. Here is an example of partition manifest:

```
{
  "name": "TFM_SP_SERVICE",
  "type": "PSA-ROT",
  "priority": "NORMAL",
  "entry_point": "tfm_service_entry",
  "stack_size": "0x1800",
  "heap_size": "0x1000",
  ...
}
```

Tooling would do manipulation to tell SPM the partition entry as '__sprtmain', and TF-M SPM would maintain the local storage at run time. Finally, the partition entry point gets called and run, tooling helps on the decoupling of SPM and SPRTL implementation. The pseudo code of a tooling result:

```
struct partition_t sp1 {
  .name = "TFM_SP_SERVICE",
  .type = PSA_ROT,
  .priority = NORMAL,
  .id = 0x00000100,
  .entry_point = __sprtmain, /* Tell SPM entry is '__sprtmain' */
  .local_storage = { /* struct sprt_local_storage_t */
    .heap_sa = sp1_heap_buf,
    .heap_sz = sizeof(sp1_heap_buf),
    .thread_entry = sp1_entry, /* Actual Partition Entry */
    .heap_instance = NULL,
  },
}
```

Implementation

The SPRTL C Runtime sources are put under: '$TFM_ROOT/secure_fw/partitions/lib/runtime/'

The output of this folder is a static library named as 'libtfm_sprt.a'. The code of 'libtfm_sprt.a' is put into a dedicated section so that a hardware protected region can be applied to contain it.

The RoT Service API are put under service interface folder. These APIs are marked with the same section attribute where 'libtfm_sprt.a' is put.

The Formatting API - 'printf' and variants

The 'printf' and its variants need special parameters passing mechanism. To implement these APIs, the toolchain provided builtin macro 'va_list', 'va_start' and 'va_end' cannot be avoided. This is because of some scenarios such as when 'stack canaries' are enabled, only the compiler knows the format of the 'canary' in order to extract the parameters correctly.

To provide a simple implementation, the following requirements are defined for 'printf':

- Format keyword 'xXduscp' needs to be supported.
- Take '%' as escape flag, '%%' shows a '%' in the formatted string.
- To save heap usage, 32 bytes buffer in the stack for collecting formatted string.
- Flush string outputting due to: a) buffer full b) function ends.

The interface for flushing can be a logging device.

Function needs implied inputs

Take 'malloc' as an example. There is only one parameter for 'malloc' in the prototype. Heap management code is put in the SPRTL for sharing with caller partitions. The heap instance belongs to each partition, which means this instance needs to be passed into the heap management code as a parameter. For allocation API in heap management, it needs two parameters - 'size' and 'instance', while for 'malloc' caller it needs a 'malloc' with one parameter 'size' only. As mentioned in the upper chapter, this instance can be retrieved from the Secure Partition Local Storage. The implementation can be:

```
void *malloc(size_t sz)
{
    struct p_sp_local_storage_t *m =
        (struct p_sp_local_storage_t *)tfm_sprt_local_storage;

    return tfm_sprt_alloc(m->heap_instance, sz);
}
```

11.3.3 TF-M Inter-Process Communication

Authors
Ken Liu, Mingyang Sun

Organization
Arm Limited

Contact
ken.liu@arm.com, mingyang.sun@arm.com

Terminology

IPC - Inter-Process Communication

For more terminology please check *Reference* document.

Design Overview

Components for implementing IPC:

- SPM – for partition information and isolation actions
- Core – for exception handling
- Memory pool
- Message manager
- Thread
- Synchronization objects
- PSA API

Implementation Details

Listed modules are all internal modules except PSA API. Prototypes and definitions are not listed for internal modules in this document. For PSA API definitions, check them in PSA Firmware Framework specification in the reference chapter.

SPM and Core

SPM manages Secure Partition information. Enhancements need to be done in SPM data structure for Secure Partition for IPC due to:

- IPC model requires each Secure Partition has its own stack area.
- Multiple services are holding in same Secure Partition and each service has its own information like message queue, SID and priority.
- Changed information related manifest items need to be changed, too.

Modifications in Core:

- More SVC calls need to be added into list since PSA API are implemented as SVC calls in TF-M.
- New PendSV handler for thread scheduling.
- Arch-related context stacking and switching.

Memory Pool

Handles of connection and messages for Secure Partition needs to be allocated dynamically. A memory pool is provided in the system to handle dynamic allocation. Each memory pool item contains below information:

- A list iterator to chain all of memory pool items.
- An information member to record information like size and types.
- The memory item body for caller usage.

A memory area needs to be provided in SPM for the memory pool. It could be an array of memory areas defined in the linker script. Two chains are available to manage the items: free chain and used chain. And an LRU (Last recent used) mechanism is applied for fast seeking while item allocating and destroying.

Message Manager

Message Manager handles message creating, pushing, retrieving and destroy. A message contains below information:

- Message sender and destination
- Message status
- IO vectors for service
- 'psa_msg_t' for service

A checking needs to be performed in SPM before creating a message to detect if a message with the same sender and destination is ongoing. This avoids repeat messages are available in the queue.

Thread

Each Secure Partition has a thread as execution environment. Secure Partition is defined statically in TF-M manifest, which indicates that a number of threads are statically defined. Threads are chained in SPM and sorted with its priority, and there is an extra indicator point to first running thread with the highest priority. This helps fast seeking of running threads while the scheduler is switching threads.

Thread context contains below information:

- Priority
- Status
- Stack pointer
- Stack pointer limitation
- Entry
- Parameter
- Entry return value
- Context
- List iterator

Thread API provides below functions:

- Thread creating and destroying
- Thread status retrieving and changing
- Current thread retrieving
- Thread context switching

PendSV exception in TF-M core is the place thread context APIs been called. Before thread switching taking place, isolation status needs to be changed based on Secure Partition change and current isolation level – a thread is a member of partition which means thread switching caused a partition switching.

Synchronization API

A first synchronization object is an event. This could be applied into event waiting in the partition, and message response handling in IPC. The event object contains below members:

- Owner thread who is waiting for this event
- Event status (Ready or Not-Ready)
- List iterator for synchronization objects management

Event API Limitation: could be waited by one thread only.

PSA API

This chapter describes the PSA API in an implementation manner.

- API type: could be Client API and Service Partition API
- Block-able: Block-able API may block caller thread; Non-Block API does not block caller thread.
- Description: The functionality description and important comments.

```
uint32_t psa_framework_version(void);
uint32_t psa_version(uint32_t sid);
```

- Client API
- Non-Block API
- These 2 functions are finally handled in SPM and return the framework version or version to the caller.

```
psa_handle_t psa_connect(uint32_t sid, uint32_t version);
psa_status_t psa_call(psa_handle_t handle, int32_t type,
                      const psa_invec *in_vec, size_t in_len,
                      psa_outvec *out_vec, size_t out_len);
void psa_close(psa_handle_t handle);
```

- Client API
- Block-able API
- These 3 APIs are implemented in the same manner and just different parameters. SPM converts each call into a corresponding message with a parameter in the message body and pushes the message into service queue to wait for the response. Scheduler switches to a specified thread (partition) and makes Secure Partition to have chance retrieving and process message. After a message response is returned to the caller, the waiting caller gets to go and get the result.

```
psa_signal_t psa_wait(psa_signal_t signal_mask, uint32_t timeout);
```

- Secure Partition API
- Block-able API
- This API blocks caller partition if there is no expected event for it. This function is implemented based on event API.

```
void psa_set_rhandle(psa_handle_t msg_handle, void *rhandle);
psa_status_t psa_get(psa_signal_t signal, psa_msg_t *msg);
size_t psa_read(psa_handle_t msg_handle, uint32_t invec_idx,
                void *buffer, size_t num_bytes);
size_t psa_skip(psa_handle_t msg_handle, uint32_t invec_idx,
                size_t num_bytes);
void psa_write(psa_handle_t msg_handle, uint32_t outvec_idx,
               const void *buffer, size_t num_bytes);
void psa_reply(psa_handle_t msg_handle, psa_status_t status);
void psa_clear(void);
void psa_eoi(psa_signal_t irq_signal);
```

- Secure Partition API
- Non-Block

- These APIs do not take the initiative to change caller status. They process data and return the processed data back to the caller.

```
void psa_notify(int32_t partition_id);
```

- Secure Partition API
- Non-Block
- This API sets DOORBELL bit in destination partition's event. This API does not take the initiative to change caller status.

```
void psa_panic(void);
```

- Secure Partition API
- Block-able API
- This function will terminate execution within the calling Secure Partition and will not return.

Reference

PSA Firmware Framework specification URL

11.3.4 Stateless Root of Trust Services Reference

Author
Mingyang Sun

Organization
Arm Limited

Contact
mingyang.sun@arm.com

Introduction

This document describes the implementation for the FF-M v1.1 feature - 'Stateless RoT Service', and the related references when developing RoT services.

It is recommended to refer to the FF-M v1.0 specification[1] and FF-M v1.1 extension[2] for background and rationale details.

[1] FF-M v1.0 Specification

[2] FF-M v1.1 Extension

Implementation Details

This chapter describes the implementation-defined items, including stateless handle value definition, tooling update, and programming API changes.

Stateless Handle Value Definition

The index, stateless indicator, and service version information are encoded into a handle by the manifest tooling, and then generated to header file `sid.h`.

Table 42:: Bit Fields of Stateless Handle

Bits	Field Description
bit 31	reserved
bit 30	stateless handle indicator bit, always 1
bit 29 - bit 16	reserved
bit 15 - bit 8	service version requested by client - for client version check
bit 7 - bit 5	reserved
bit 4 - bit 0	the handle index, [0, 31]

Since connection-based services and stateless services share the same PSA API `psa_call()`, an indicator bit is set in the handle indicate the type of the handle. If it is set, the handle is stateless, and definition is as described in the table above. Maximum connection-based handle is 0x3FFFFFFF, thus the indicator bit is always 0.

The index is used for organizing stateless services in manifest tool and locating a stateless service in SPM logic. A range of index [0, 31] is the initial implementation. Future expansion is possible.

Tooling Support

TF-M provides a tool (`tools/tfm_parse_manifest_list.py`) to generate source header files required by partition and services. For example, the generated `sid.h` contains service ID and version. The tooling is extended to generate stateless handle from partition manifests automatically.

The `stateless_handle` attribute in manifest is only supported by partitions with firmware framework version 1.1.

- If `stateless_handle` in manifest is set to an integer, the index is `stateless_handle - 1`.
- If it is `auto` or not set, the first empty index in range [0, 31] is assigned.
- Other settings - tooling reports an error.

Finally, the tooling encodes the handle according to definitions in *Stateless Handle Value Definition* section, and writes them to `sid.h` header file.

Changes in Programming API

This chapter describes the changes in programming API for stateless services. The following APIs' behavious and message data structure members are updated to support the stateless service.

psa_connect()

According to FF-M v1.1, client calling psa_connect() with the SID of a stateless RoT Service is a PROGRAMMER_ERROR.

psa_close()

According to FF-M v1.1, client passing a stateless handle to call this API is a PROGRAMMER_ERROR.

psa_call()

```
psa_status_t psa_call(psa_handle_t handle, int32_t type,
                      const psa_invec *in_vec, size_t in_len,
                      psa_outvec *out_vec, size_t out_len)
```

API parameters and behaviour change:

1. The handle parameter must be a stateless handle defined in psa_manifest/sid.h when requesting a stateless service.
2. This API validates stateless handle, decodes index and service version information from it. SPM uses the index to know which stateless service is requested.
3. This API performs some operations as psa_connect() does, such as the authorization check, service and client version check, and handle space allocation.

Service behaviour change during a “psa_call”:

Service does not accept connection and disconnection messages. After a “psa_call” request is serviced, it calls psa_reply(), frees the connection handle to handle pool.

psa_set_rhandle()

According to FF-M v1.1, stateless service calling this API on a message is a PROGRAMMER_ERROR and it will never return.

psa_msg_t type

The rhandle member of a psa_msg_t type received by a stateless RoT Service is always NULL.

Application Recommendation

There are particular services that do not need sessions. The access to the service is a one-shot connection. These services provide standalone operations that do not require any non-volatile state of resources to be maintained by the RoT service itself or do not expose any kind of context or session to the caller. Such services are recommended to be implemented as stateless, to provide quick access and to avoid extra overheads.

In this case, rhandle access would be prohibited as it is used for saving state or non-volatile resources while stateless services do not need them.

Update Feasibility of Existing Services

TF-M native services are used widely. They only need standalone operations and do not need to keep state between sessions. For example, the service in Crypto partition does not do anything during `psa_connect()` or `psa_close()` process. Same for services in other partitions, thus all of them can be implemented as stateless.

Analysis for them:

Table 43:: TF-M Partition Services Update Possibility

Partition	Number of Services	Can be Stateless
ITS	4	All
PS	5	All
Crypto	1	All
FWU	6	All
Platform	4	All
Initial Attestation	2	All

Other services are not analyzed here.

Grouping Services

Stateless service table is stored statically, and TF-M supports 32 stateless services currently.

Similar stateless services in a partition could be grouped, and assign one `SID` for the group. The `type` parameter in `psa_call()` could be extended to identify the service in group. In this case, it is recommended to use consecutive values for `type`.

It is recommended that each Secure Partition declares one stateless service and uses the type parameter to distinguish different stateless services. Therefore, more stateless services can be supported.

Migrating to Stateless Services

Please refer to Chapter 4 “Stateless Root of Trust services”, Appendix B.3.2 “Using a stateless RoT Service”, and Appendix D “Implementing session-less RoT Services” in FF-M v1.1 document for details on which kind of service can be stateless and how to implement a stateless service.

Reference

11.3.5 Uniform Secure Service Signature

Author
Miklos Balint

Organization
Arm Limited

Contact
Miklos Balint <miklos.balint@arm.com>

Declaring secure service interface

The following alternative secure service signature is proposed as an amendment to existing implementation.

Individual signatures - current method

A `<service_name>_veneers.c` file is created in the `secure_fw/ns_callable` directory, that specifies the signature for each veneer function, and calls the secure function from the veneers. The respective `interface/include/<service_name>_veneers.h` file with the veneer declarations have to be created and maintained manually. Note that at present TF-M framework limits the range of valid return values a secure service can provide, reserving a range for framework error codes.

Uniform signatures - proposal

The proposal is to use a uniform signature for all the secure functions of the secure service. There are multiple advantages of this method:

- TF-M Core can do a sanity check on the access rights of the veneer parameters, and there is no need for the secure services to make these checks individually. Please note that in the present implementation sanity check is only fully supported for level 1 isolation.
- The veneer declarations and implementations for the secure functions can be generated automatically from a template (using the secure function list in the secure service's manifest)

The signature for such secure services would look like this:

```
psa_status_t secure_function_name(struct psa_invec *in_vec, size_t in_len,
                                  struct psa_outvec *out_vec, size_t out_len);
```

where

Return value:

`psa_status_t` is a status code whose values are described in PSA Firmware Framework (as in version 1.0-beta-0 chapter 4.3.3).

Note:

The return value limitations imposed by TF-M framework for proprietary secure service veneers would not apply to secure services using the uniform signature. This is analogous to how PSA Firmware Framework handles values returned by `psa_reply()` function.

Arguments:

```
/**
 * A read-only input memory region provided to a RoT Service.
 */
typedef struct psa_invec {
    const void *base;   /*!< the start address of the memory buffer */
    size_t len;         /*!< the size in bytes */
} psa_invec;

/**
 * A writable output memory region provided to a RoT Service.
 */
typedef struct psa_outvec {
    void *base;         /*!< the start address of the memory buffer */
    size_t len;         /*!< the size in bytes */
} psa_outvec;

/**
 * in_len: the number of input parameters, i.e. psa_invecs
 * out_len: the number of output parameters, i.e. psa_outvecs
 */
```

The number of vectors that can be passed to a secure service is constrained:

```
in_len + out_len <= PSA_MAX_IOVEC
```

The veneer function declarations and implementations are generated in the `interface/include/tfm_veneers.h` and `secure_fw\ns_callable\tfm_veneers.c` files respectively. The veneer functions are created with the name `tfm_<secure_function_name>_veneer`

Services that implement the uniform signature do not need to manually fill the template veneer function to call `TFM_CORE_SFN_REQUEST` macro.

Compatibility

Note that the proposal is for the two types of services (those with proprietary signatures and those with uniform signatures) to co-exist, with the intention of eventually phasing out proprietary signatures in favour of the more robust, uniform signature.

11.3.6 Crypto Service design

Author
Antonio de Angelis

Organization
Arm Limited

Contact
Antonio de Angelis <antonio.deangelis@arm.com>

Table of Contents

Abstract

This document describes the design of the TF-M Cryptographic Secure Service (in short, TF-M Crypto service).

Introduction

The TF-M Crypto service provides an implementation of the PSA Crypto API in a PSA RoT secure partition in TF-M. It is based on Mbed Crypto, which is a reference implementation of the PSA Crypto API. For more details on the PSA Crypto API or the Mbed Crypto implementation, please refer directly to the `mbed-crypto` GitHub repository[1] .

The service can be used by other services running in the SPE, or by applications running in the NSPE, to provide cryptographic functionalities.

Components

The TF-M Crypto service is implemented by a number of different software components, which are listed below:

[1] `mbed-crypto` repository which holds the PSA Crypto API specification and the Mbed Crypto reference implementation: https://github.com/Mbed-TLS

Table 44:: Components table

Component name	Description	Location
Client API interface	This module exports the client API of PSA Crypto to the users.	`./interface/src/tfm_crypto_api.c`
Mbed Crypto	The Mbed Crypto library is used in the service as a cryptographic library exposing the PSA Crypto API interface.	Needed as dependency at the same level of the TF-M folder, i.e. `../mbed-crypto`
Init module	This module handles the initialisation of the service objects during TF-M boot and provides the infrastructure to service requests when TF-M is built for IPC model. The dispatching mechanism of IPC requests is based on a look up table of function pointers. This design allows for better scalability and support of a higher number of Secure functions with minimal overhead and duplication of code.	`./secure_fw/partitions/crypto/crypto_init.c`
Alloc module	This module handles the allocation of contexts for multipart operations in the Secure world.	`./secure_fw/partitions/crypto/crypto_alloc.c`
Service modules	These modules (AEAD, Asymmetric, Cipher, Key Deriv, Hash, Key, MAC) represent a thin layer which is in charge of servicing the calls from the SPE/NSPE client API interfaces. They provide parameter sanitation and context retrieval for multipart operations, and dispatching to the corresponding library function exposed by Mbed Crypto for the desired functionality.	`./secure_fw/partitions/crypto/crypto_aead.c` `./secure_fw/partitions/crypto/crypto_asymmetric.c` `./secure_fw/partitions/crypto/crypto_cipher.c` `./secure_fw/partitions/crypto/crypto_key_derivation.c` `./secure_fw/partitions/crypto/crypto_hash.c` `./secure_fw/partitions/crypto/crypto_key.c` `./secure_fw/partitions/crypto/crypto_mac.c` ‘‘./secure_fw/partitions/crypto/crypto_key_managemen
Manifest	The manifest file is a description of the service components.	`./secure_fw/partitions/crypto/manifest.yaml`
CMake files and headers	The CMake files are used by the TF-M CMake build system to build the service as part of the Secure FW build. The service is built as a static library (`tfm_crypto.a`). The build system allows to build as well the Mbed Crypto library as part of the Secure FW build process and archive it with the static library of the Crypto service. The headers are used to export the public prototypes of the functions in the Service modules `tfm_crypto_api.h`, and to provide the necessary defines (i.e. `TFM_CRYPTO_SIG`). In particular `TFM_CRYPTO_SIG` identifies the signal on which the service handler waits for requests when the service is built for IPC model. The header available in the interface, `tfm_crypto_defs.h` , contains types and defines for building the NSPE interface as part of a Non-Secure application. Finally, the `crypto_spe.h` header is used during the build of the Mbed Crypto library, when the Mbed Crypto config option `MBEDTLS_PSA_CRYPTO_SPM` is defined, to add a custom prefix to the PSA API symbols so that duplication of symbol names is avoided. The prefix used for the PSA API symbols	`./secure_fw/partitions/crypto/CMakeLists.inc` `./secure_fw/partitions/crypto/CMakeLists.txt` `./interface/include/tfm_crypto_defs.h` `./secure_fw/partitions/crypto/tfm_crypto_api.h` `./secure_fw/partitions/crypto/tfm_crypto_signal.h` `./secure_fw/partitions/crypto/spe_crypto.h`

The interaction between the different components is described by the following block diagram:

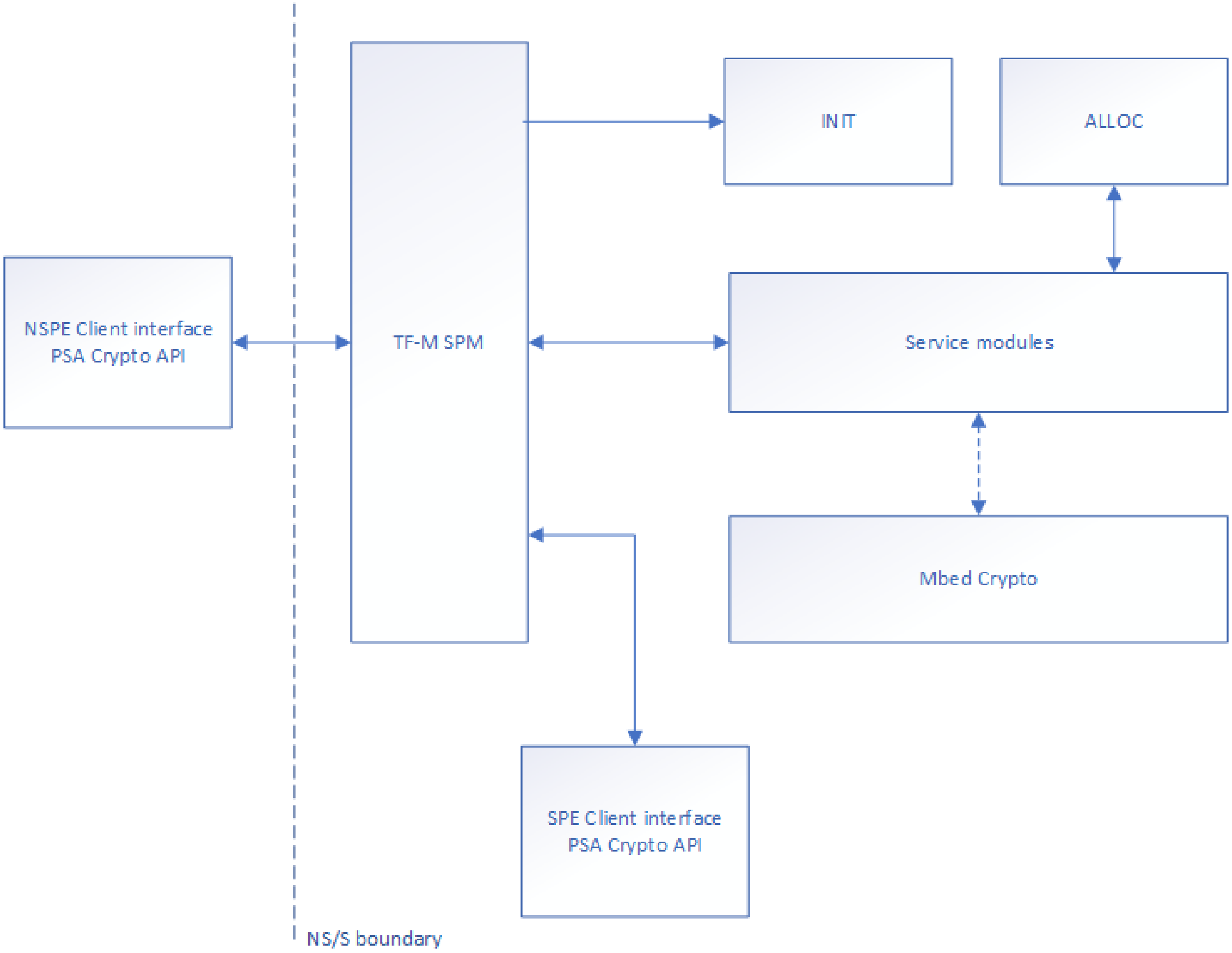


Figure 12:: Block diagram of the different components of the TF-M Crypto service. A dotted line is used to indicate the interaction with a library.

Note: in IPC model, the interaction between components is slightly different, as the Service modules are not called directly through the TF-M Secure Partition Manager but through the IPC handler which resides in the Init module.

Service API description

Most of the APIs exported by the TF-M Crypto service (i.e. from the Service modules) have a direct correspondence with the PSA Crypto API. The Alloc and Init modules instead export some APIs which are specific to the TF-M Crypto service, and are available only to the Service modules or the SPM. For a detailed description of the prototypes please refer to the `tfm_crypto_api.h` header.

Table 45:: Init and Alloc modules APIs

Function	Module	Caller	Scope
tfm_crypto_init()	Init	SPM	Called during TF-M boot for initialisation. In IPC model, it calls the IPC service request handler.
tfm_crypto_init_alloc()	Alloc	Init	Called by tfm_crypto_init(), it initialises the concurrent operation contexts storage area.
tfm_crypto_operation_alloc()	Alloc	Service modules	It allocates a new operation context for a multipart operation. It returns an handle to the allocated context in secure memory.
tfm_crypto_operation_lookup()	Alloc	Service modules	It retrieves a previously allocated operation context of a multipart operation, based on the handle given as input.
tfm_crypto_operation_release()	Alloc	Service modules	It releases a previously allocated operation context of a multipart operation, based on the handle given as input.

Configuration parameters

The TF-M Crypto service exposes some configuration parameters to tailor the service configuration in terms of supported functionalities and hence FLASH/RAM size to meet the requirements of different platforms and use cases. These parameters can be provided via CMake parameters during the CMake configuration step and as a configuration header to allow the configuration of the Mbed Crypto library.

Table 46:: Configuration parameters table

Parameter	Type	Description	Scope	Default
CRYPTO_ENGINE_BUF_SIZE	CMake build configuration parameter	Buffer used by Mbed Crypto for its own allocations at runtime. This is a buffer allocated in static memory.	To be configured based on the desired use case and application requirements.	8096 (bytes)
CRYPTO_CONC_OPER_NUM	CMake build configuration parameter	This parameter defines the maximum number of possible concurrent operation contexts (cipher, MAC, hash and key deriv) for multi-part operations, that can be allocated simultaneously at any time.	To be configured based on the desire use case and platform requirements.	8
CRYPTO_IOVEC_BUFFER_SIZE	CMake build configuration parameter	This parameter applies only to IPC model builds. In IPC model, during a Service call, input and outputs are allocated temporarily in an internal scratch buffer whose size is determined by this parameter.	To be configured based on the desired use case and application requirements.	5120 (bytes)
MBEDTLS_CONFIG_FILE	Configuration header	The Mbed Crypto library can be configured to support different algorithms through the usage of a a configuration header file at build time. This allows for tailoring FLASH/RAM requirements for different platforms and use cases.	To be configured based on the application and platform requirements.	`./lib/ext/mbedcrypto/mbedcrypto_config/tfm_mbedcrypto_config_default.h`
MBEDTLS_PSA_CRYPTO_CONFIG_FILE	Configuration header	This header file specifies which cryptographic mechanisms are available through the PSA API when #MBEDTLS_PSA_CRYPTO_CONFIG is enabled, and is not used when #MBEDTLS_PSA_CRYPTO_CONFIG is disabled.	To be configured based on the application and platform requirements.	`./lib/ext/mbedcrypto/mbedcrypto_config/crypto_config_default.h`

References

11.3.7 Symmetric key algorithm based Initial Attestation

Author
David Hu

Organization
Arm Limited

Contact
david.hu@arm.com

Introduction

This document proposes a design of symmetric key algorithm based Initial Attestation in TF-M.

Symmetric key algorithm based Initial Attestation (*symmetric Initial Attestation* for short) signs and verifies Initial Attestation Token (IAT) with a symmetric cryptography signature scheme, such as HMAC. It can reduce TF-M binary size and memory footprint on ultra-constrained devices without integrating asymmetric ciphers.

This proposal follows PSA Attestation API document[1].

Note: As pointed out by PSA Attestation API[Page 379, 1], the use cases of Initial Attestation based on symmetric key algorithms can be limited due to the associated infrastructure costs for key management and operational complexities. It may also restrict the ability to interoperate with scenarios that involve third parties.

Design overview

The symmetric Initial Attestation follows the existing IAT generation sequence for Initial Attestation based on asymmetric key algorithm (*asymmetric Initial Attestation* for short).

As Profile Small design[2] requests, a configuration flag `SYMMETRIC_INITIAL_ATTESTATION` selects symmetric initial attestation during build.

The top-level design is shown in *Overall design diagram* below.

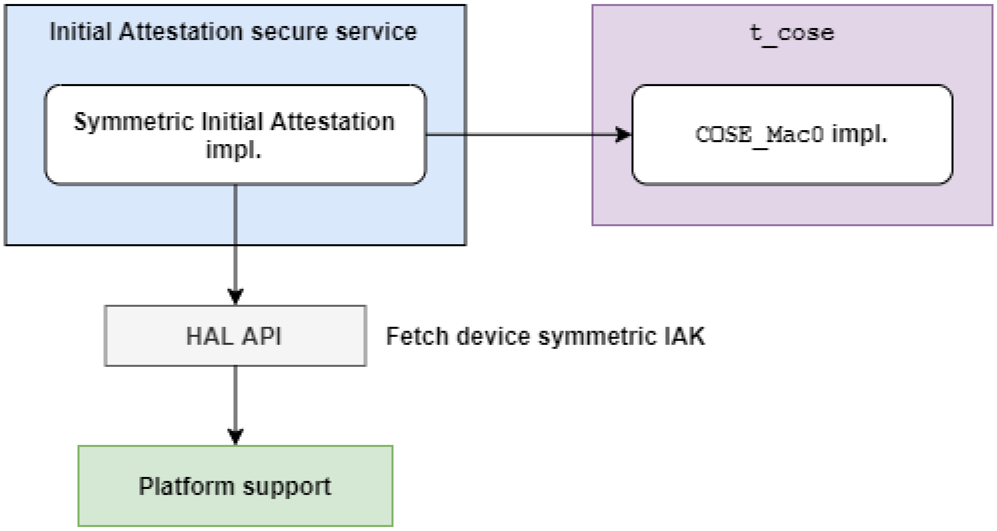

Initial Attestation secure service
Symmetric Initial Attestation impl.
t_cose
COSE_Mac0 impl.
HAL API
Fetch device symmetric IAK
Platform support

Figure 13:: Overall design diagram

Symmetric Initial Attestation adds its own implementations of some steps in IAT generation in Initial Attestation secure service. More details are covered in *IAT generation in Initial Attestation secure service*.

The interfaces and procedures of Initial Attestation secure service are not affected. Refer to Initial Attestation Service Integration Guide[3] for details of the implementation of Initial Attestation secure service.

Symmetric Initial Attestation invokes `t_cose` library to build up `COSE_Mac0` structure. `COSE_Mac0` support is added to `t_cose` library in TF-M since official `t_cose` hasn't supported `COSE_Mac0` yet. The design of `COSE_Mac0` support is covered in *COSE_Mac0 support in t_cose*.

[1] PSA Attestation API 1.0 (ARM IHI 0085)
[2] *Trusted Firmware-M Profile Small Design*
[3] *Initial Attestation Service Integration Guide*

Note: The COSE_Mac0 implementation in this proposal is a prototype only for Proof of Concept so far. It may be replaced after t_cose officially supports COSE_Mac0 message.

Several HAL APIs are defined to fetch platform specific assets required by Symmetric Initial Attestation. For example, tfm_plat_get_symmetric_iak() fetches symmetric Initial Attestation Key (IAK). Those HAL APIs are summarized in *HAL APIs*.

Decoding and verification of symmetric Initial Attestation is also included in this proposal for regression test. The test suites and IAT decoding are discussed in *TF-M Test suite*.

QCBOR library and Crypto service are also invoked. But this proposal doesn't require any modification to either QCBOR or Crypto service. Therefore, descriptions of QCBOR and Crypto service are skipped in this document.

IAT generation in Initial Attestation secure service

The sequence of IAT generation of symmetric Initial Attestation is shown in *Symmetric IAT generation flow in Initial Attestation secure service* below. Note that the Register symmetric IAK stage is no longer required due to changes in the Crypto partition (attest_symmetric_key.c is now responsible only for calculating the instance ID).

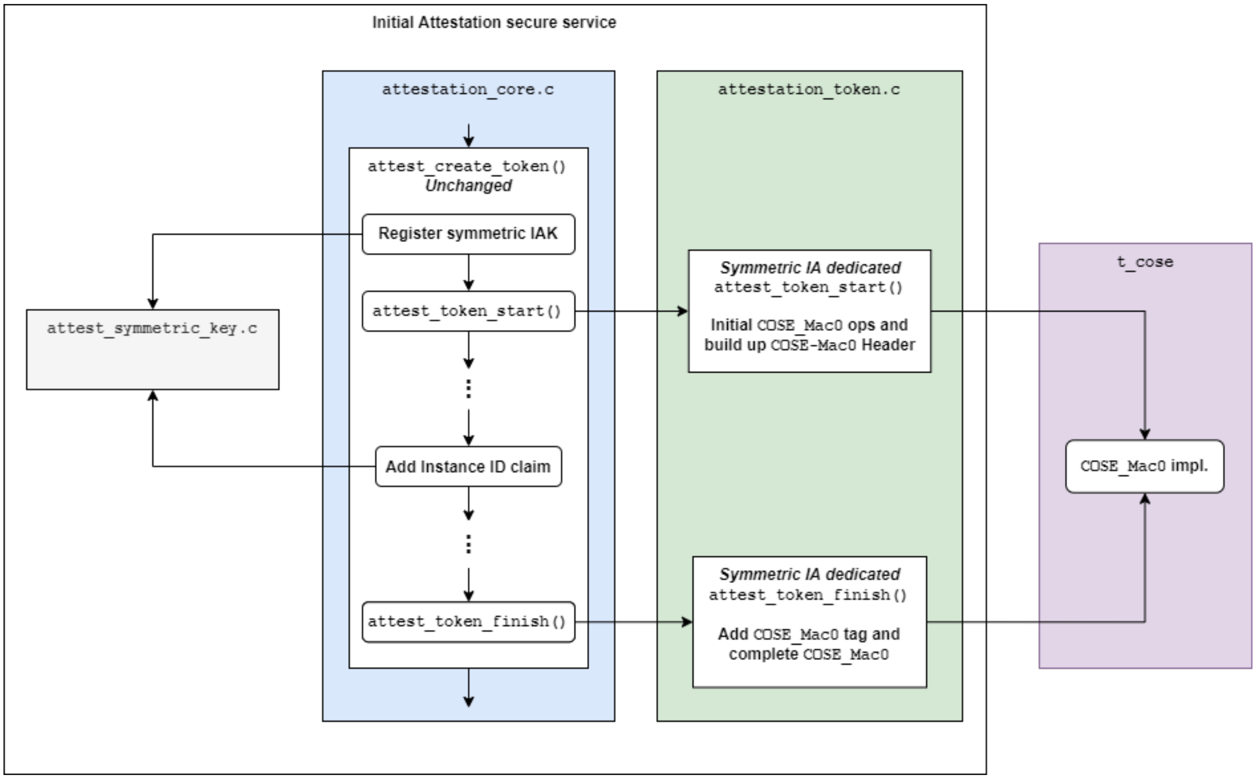


Figure 14:: Symmetric IAT generation flow in Initial Attestation secure service

In Initial Attestation secure service, symmetric Initial Attestation implements the following steps in attest_create_token(), which are different from those of asymmetric Initial Attestation.

- attest_token_start()
- Instance ID claims
- attest_token_finish()

If `SYMMETRIC_INITIAL_ATTESTATION` is selected, symmetric Initial Attestation dedicated implementations of those steps are included in build. Otherwise, asymmetric Initial Attestation dedicated implementations are included instead.

Symmetric Initial Attestation implementation resides a new file `attest_symmetric_key.c` to handle symmetric Instance ID related operations. Symmetric Initial Attestation dedicated `attest_token_start()` and `attest_token_finish()` are added in `attestation_token.c`.

The details are covered in following sections.

Symmetric Instance ID

Symmetric Initial Attestation dedicated `attest_symmetric_key.c` implements the `attest_get_instance_id()` function. This function returns the Instance ID value, calculating it if it has not already been calculated. Refer to *Instance ID claim_* for more details.

Note: Only symmetric IAK for HMAC algorithm is allowed so far.

Instance ID calculation

In symmetric Initial Attestation, Instance ID is also calculated the first time it is requested. It can protect critical symmetric IAK from being frequently fetched, which increases the risk of asset disclosure.

The Instance ID value is the output of hashing symmetric IAK raw data *twice*, as requested in PSA Attestation API[1]. HMAC-SHA256 may be hard-coded as the hash algorithm of Instance ID calculation.

Note: According to RFC2104[4], if a HMAC key is longer than the HMAC block size, the key will be first hashed. The hash output is used as the key in HMAC computation.

In current design, HMAC is used to calculate the authentication tag of `COSE_Mac0`. Assume that symmetric IAK is longer than HMAC block size (HMAC-SHA256 by default), the Instance ID is actually the HMAC key for `COSE_Mac0` authentication tag generation, if Instance ID value is the output of hashing IAK only *once*. Therefore, attackers may request an valid IAT from device and fake malicious ones by using Instance ID to calculate valid authentication tags, to cheat others.

As a result, symmetric IAK raw data should be hashed *twice* to generate the Instance ID value.

The Instance ID calculation result is stored in a static buffer. Token generation process can call `attest_get_instance_id()` to fetch the data from that static buffer.

attest_token_start()

Symmetric Initial Attestation dedicated `attest_token_start()` initializes the `COSE_Mac0` signing context and builds up the `COSE_Mac0` Header.

The workflow inside `attest_token_start()` is shown in *Workflow in symmetric Initial Attestation attest_token_start()* below.

Descriptions of each step are listed below:

1. `t_cose_mac0_sign_init()` is invoked to initialize `COSE_Mac0` signing context in `t_cose`.

[4] HMAC: Keyed-Hashing for Message Authentication

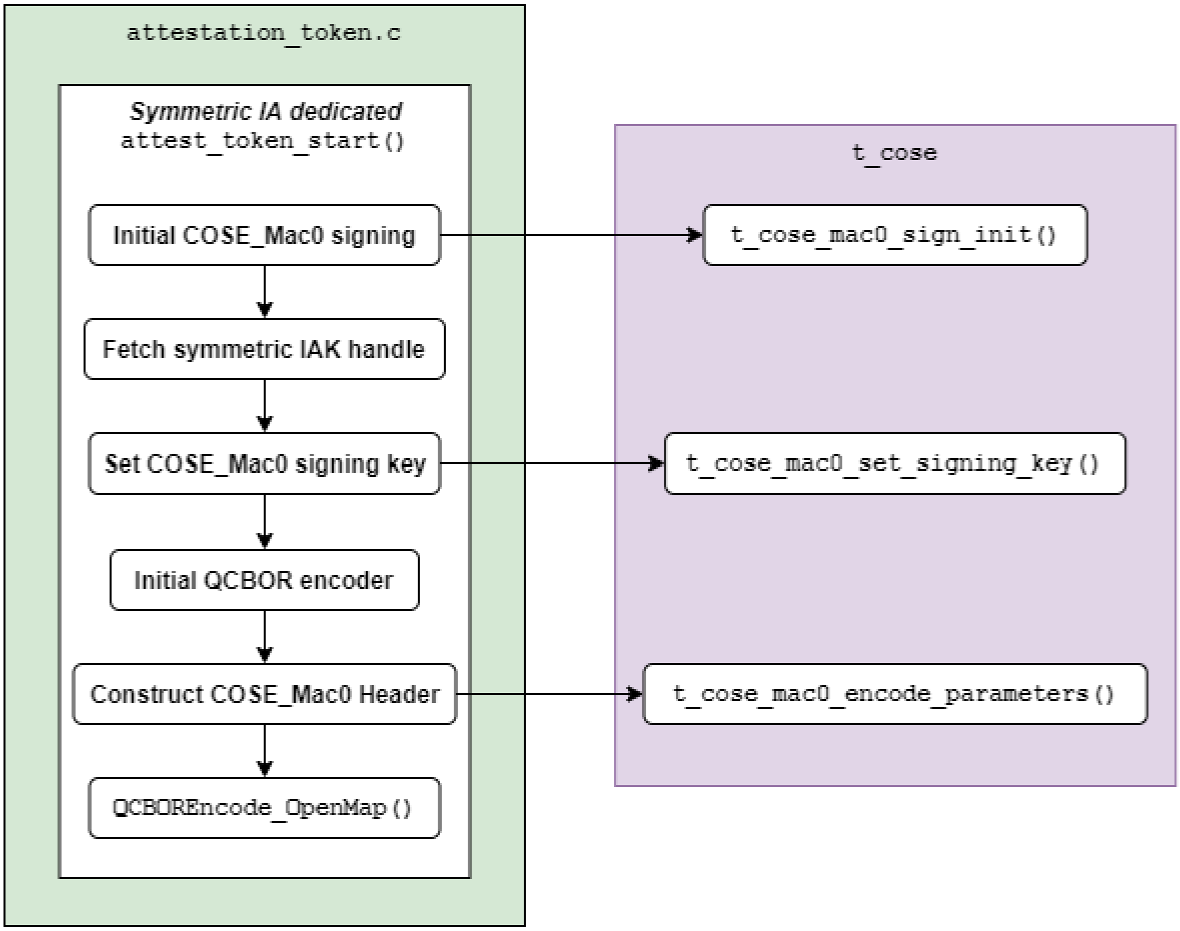

attestation_token.c
Symmetric IA dedicated
attest_token_start()
Initial COSE_Mac0 signing
Fetch symmetric IAK handle
Set COSE_Mac0 signing key
Initial QCBOR encoder
Construct COSE_Mac0 Header
QCBOREncode_OpenMap()
t_cose
t_cose_mac0_sign_init()
t_cose_mac0_set_signing_key()
t_cose_mac0_encode_parameters()

Figure 15:: Workflow in symmetric Initial Attestation `attest_token_start()`

2. The symmetric IAK handle is set into `COSE_Mac0` signing context via `t_cose_mac0_set_signing_key()`.
3. Initialize `QCBOR` encoder.
4. The header parameters are encoded into `COSE_Mac0` structure in `t_cose_mac0_encode_parameters()`.
5. `QCBOREncode_OpenMap()` prepares for encoding the `COSE_Mac0` payload, which is filled with IAT claims.

All the `COSE_Mac0` functionalities in `t_cose` are covered in *COSE_Mac0 support in t_cose*.

Instance ID claim

Symmetric Initial Attestation also implements Instance ID claims in `attest_add_instance_id_claim()`.

The Instance ID value is fetched via `attest_get_instance_id()`. The value has already been calculated during symmetric IAK registration. See *Instance ID calculation* for details.

The other steps are the same as those in asymmetric Initial Attestation implementation. The UEID type byte is set to 0x01.

attest_token_finish()

Symmetric Initial Attestation dedicated `attest_token_finish()` calls `t_cose_mac0_encode_tag()` to calculate and encode the authentication tag of `COSE_Mac0` structure.

The whole COSE and CBOR encoding are completed in `attest_token_finish()`.

The simplified flow in `attest_token_finish()` is shown in *Workflow in symmetric Initial Attestation attest_token_finish()* below.

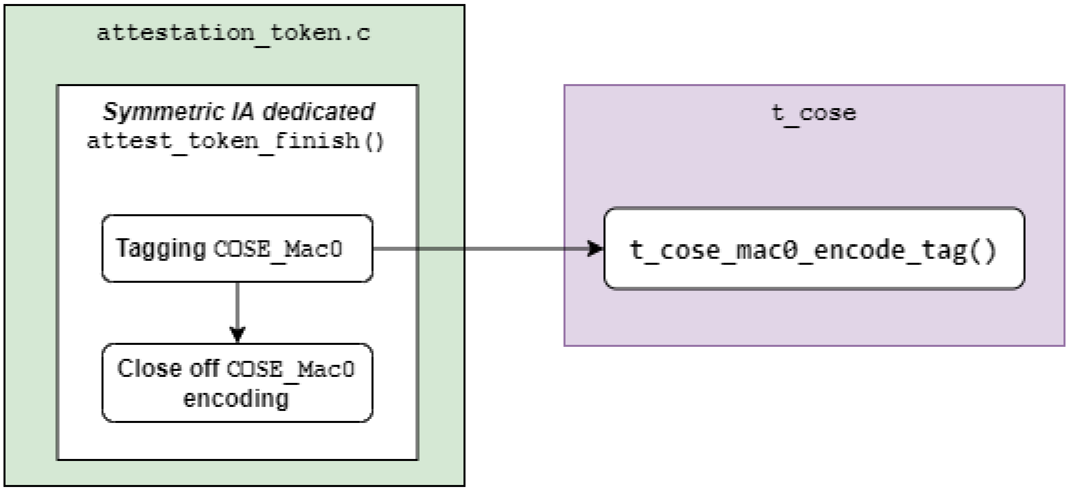


Figure 16:: Workflow in symmetric Initial Attestation `attest_token_finish()`

COSE_Mac0 support in t_cose

COSE_Mac0 supports in t_cose in TF-M include the following major functionalities:

- Encoding COSE_Mac0 structure
- Decoding and verifying COSE_Mac0 structure
- HMAC computation to generate and verify authentication tag
- Short-circuit tagging for test mode

According to RFC8152[5], COSE_Mac0 and COSE_Sign1 have similar structures. Therefore, the prototype follows COSE_Sign1 implementation to build up COSE_Mac0 file structure and implement COSE_Mac0 encoding and decoding.

Although COSE_Mac0 can share lots of data types, APIs and encoding/decoding steps with COSE_Sign1 in implementation, this prototype separates COSE_Mac0 implementation from COSE_Sign1. COSE_Mac0 owns its dedicated signing/verification contexts, APIs and encoding/decoding process. The purposes of separating COSE_Mac0 and COSE_Sign1 are listed below

- It can keep changes to COSE_Sign1 as small as possible and avoid conflicts with development in COSE_Sign1`. It can decrease conflicts if t_cose in TF-M is synchronized with original t_cose repository later.
- COSE_Mac0 and COSE_Sign1 are exclusive in TF-M use cases. It cannot decrease TF-M memory footprint by extracting the common components shared by COSE_Mac0 and COSE_Sign1 but can make the design over-complicated.

Note: Only HMAC is supported in current COSE_Mac0 prototype.

File structure

New files are added to implement the functionalities listed above. The structure of files is shown in the table below.

Table 47:: New files in t_cose

Directory	Files	Descriptions
src	t_cose_mac0_sign.c	Encode COSE_Mac0 structure
	t_cose_mac0_verify.c	Decode and verify COSE_Mac0 structure.
inc	t_cose_mac0_sign.h	Data type definitions and function declarations of encoding and signing COSE_Mac0 message.
	t_cose_mac0_verify.h	Data type definitions and function declarations of verifying COSE_Mac0 message.

Other t_cose files may also be changed to add COSE_Mac0 associated data types and function declarations.

HMAC operations are added in crypto_adapters/t_cose_psa_crypto.c. Preprocessor flags are added to select corresponding crypto for COSE message signing and verification.

- T_COSE_ENABLE_SIGN1 selects ECDSA and Hash operations for COSE_Sign1.
- T_COSE_ENABLE_MAC0 selects HMAC operations for COSE_Mac0.

[5] CBOR Object Signing and Encryption (COSE)

Encoding COSE_Mac0

Following COSE_Sign1 implementation, COSE_Mac0 encoding exports similar functions to Initial Attestation secure service. The major functions are listed below.

Initialize signing context

t_cose_mac0_sign_init() initializes COSE_Mac0 signing context and configures option flags and algorithm used in signing.

```
static void
t_cose_mac0_sign_init(struct t_cose_mac0_sign_ctx *me,
                      int32_t                      option_flags,
                      int32_t                      cose_algorithm_id);
```

The COSE_Mac0 signing context is defined as

```
struct t_cose_mac0_sign_ctx {
    /* Private data structure */
    uint8_t               protected_parameters_buffer[
                                T_COSE_MAC0_MAX_SIZE_PROTECTED_PARAMETERS];
    struct q_useful_buf_c protected_parameters; /* The encoded protected parameters */
    int32_t               cose_algorithm_id;
    struct t_cose_key     signing_key;
    int32_t               option_flags;
    struct q_useful_buf_c kid;
    ...
};
```

Set signing key

t_cose_mac0_set_signing_key() sets the key used in COSE_Mac0 signing. Optional kid, as a key identifer, will be encoded into COSE_Mac0 Header unprotected bucket.

```
static void
t_cose_mac0_set_signing_key(struct t_cose_mac0_sign_ctx *me,
                            struct t_cose_key            signing_key,
                            struct q_useful_buf_c        kid);
```

Encode Header parameters

t_cose_mac0_encode_parameters() encodes the COSE_Mac0 Header parameters and outputs the encoded context to cbor_encode_ctx.

```
enum t_cose_err_t
t_cose_mac0_encode_parameters(struct t_cose_mac0_sign_ctx *context,
                              QCBOREncodeContext          *cbor_encode_ctx);
```

Calculate and add authentication tag

`t_cose_mac0_encode_tag()` calculates the authentication tag and finishes the `COSE_Mac0` message.

```
enum t_cose_err_t
t_cose_mac0_encode_tag(struct t_cose_mac0_sign_ctx *context,
                       QCBOREncodeContext          *cbor_encode_ctx);
```

Decoding COSE_Mac0

Following `COSE_Sign1` implementation, `COSE_Mac0` decoding exports similar functions to test suite of Initial Attestation. The major functions are listed below.

Initialize verification context

`t_cose_mac0_verify_init()` initializes `COSE_Mac0` verification context and configures option flags in verification.

```
static void
t_cose_mac0_verify_init(struct t_cose_mac0_verify_ctx *context,
                        int32_t                        option_flags);
```

The `COSE_Mac0` verification context is defined as

```
struct t_cose_mac0_verify_ctx {
    /* Private data structure */
    struct t_cose_key     verification_key;
    int32_t               option_flags;
};
```

Set verification key

`t_cose_mac0_set_verify_key()` sets the key for verifying `COSE_Mac0` authentication tag.

```
static void
t_cose_mac0_set_verify_key(struct t_cose_mac0_verify_ctx *context,
                           struct t_cose_key              verify_key);
```

Decode and verify COSE_Mac0

`t_cose_mac0_verify()` decodes the `COSE_Mac0` structure and verifies the authentication tag.

```
enum t_cose_err_t
t_cose_mac0_verify(struct t_cose_mac0_verify_ctx *context,
                   struct q_useful_buf_c          cose_mac0,
                   struct q_useful_buf_c         *payload,
                   struct t_cose_parameters      *parameters);
```

Short-circuit tagging

If T_COSE_OPT_SHORT_CIRCUIT_TAG option is enabled, COSE_Mac0 encoding will hash the COSE_Mac0 content and add the hash output as an authentication tag. It is useful when critical symmetric IAK is unavailable or cannot be accessed, perhaps because it has not been provisioned or configured for the particular device. It is only for test and must not be used in actual use case. The kid parameter will either be skipped in COSE_Mac0 Header.

If T_COSE_OPT_ALLOW_SHORT_CIRCUIT option is enabled, COSE_Mac0 decoding will only verify the hash output, without requiring symmetric key for authentication tag verification.

TF-M Test suite

Symmetric Initial Attestation adds dedicated non-secure and secure test suites. The test suites also follow asymmetric Initial Attestation test suites implementation but optimize the memory footprint. Symmetric Initial Attestation non-secure and secure test suites request Initial Attestation secure service to generate IATs. After IATs are generated successfully, test suites decode IATs and parse the claims. Secure test suite also verifies the authentication tag in COSE_Mac0 structure.

Symmetric Initial Attestation implements its dedicated attest_token_decode_validate_token() in attest_symmetric_iat_decoded.c to perform IAT decoding required by test suites. If SYMMETRIC_INITIAL_ATTESTATION is selected, attest_symmetric_iat_decoded.c is included in build. Otherwise, asymmetric Initial Attestation dedicated implementations are included instead.

The workflow of symmetric Initial Attestation dedicated attest_token_decode_validate_token() is shown below.

If the decoding is required from secure test suite, attest_token_decode_validate_token() will fetch symmetric IAK to verify the authentication tag in COSE_Mac0 structure. If the decoding is required from non-secure test suite, attest_token_decode_validate_token() will decode COSE_Mac0 only by setting T_COSE_OPT_DECODE_ONLY option flag. Non-secure must not access the symmetric IAK.

HAL APIs

HAL APIs are summarized below.

Fetch device symmetric IAK

tfm_plat_get_symmetric_iak() fetches device symmetric IAK.

```
enum tfm_plat_err_t tfm_plat_get_symmetric_iak(uint8_t *key_buf,
                                               size_t buf_len,
                                               size_t *key_len,
                                               psa_algorithm_t *key_alg);
```

Parameters:

key_buf	Buffer to store the symmetric IAK.
buf_len	The length of key_buf.
key_len	The length of the symmetric IAK.
key_alg	The key algorithm. Only HMAC SHA-256 is supported so far.

It returns error code specified in enum tfm_plat_err_t.

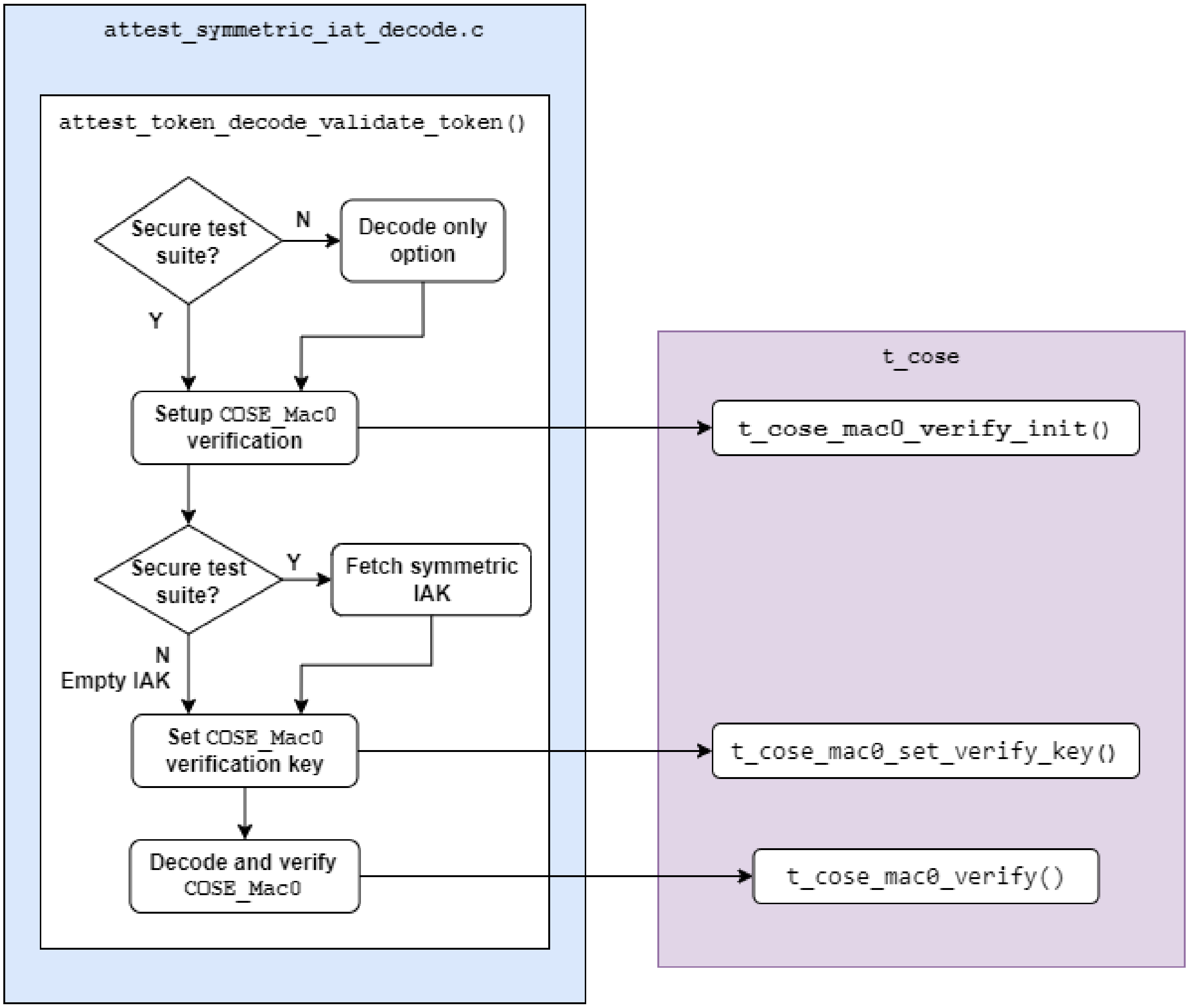

attest_symmetric_iat_decode.c
attest_token_decode_validate_token()
Secure test suite?
N
Decode only option
Y
Setup COSE_Mac0 verification
Secure test suite?
Y
Fetch symmetric IAK
N
Empty IAK
Set COSE_Mac0 verification key
Decode and verify COSE_Mac0
t_cose
t_cose_mac0_verify_init()
t_cose_mac0_set_verify_key()
t_cose_mac0_verify()

Figure 17:: Workflow in symmetric Initial Attestation `attest_token_decode_validate_token()`

Get symmetric IAK key identifier

attest_plat_get_symmetric_iak_id() gets the key identifier of the symmetric IAK as the kid parameter in COSE Header.

Optional if device doesn't install a key identifier for symmetric IAK.

```
enum tfm_plat_err_t attest_plat_get_symmetric_iak_id(void *kid_buf,
                                                     size_t buf_len,
                                                     size_t *kid_len);
```

Parameters:

kid_buf	Buffer to store the IAK identifier.
buf_len	The length of kid_buf.
kid_len	The length of the IAK identifier.

It returns error code specified in enum tfm_plat_err_t.

Reference

11.3.8 Internal Trusted Storage (ITS) Service

Author
Jamie Fox

Organization
Arm Limited

Contact
Jamie Fox <jamie.fox@arm.com>

Add support for block-aligned flash in Internal Trusted Storage

Author
Minos Galanakis

Organization
Arm Limited

Contact
Minos Galanakis <minos.galanakis@arm.com>

Abstract

The proposal is describing a mechanism to enable the use of larger flash devices, imposing a requirement for word-aligned full-block program operations, in Trusted Firmware-M.

Requirements

- Allow page-aligned writes for up to 512 Bytes per page.
- Guarantee data integrity and power-failure reliability.
- Do not alter existing supported platform behaviour.

Current implementation

In the current ITS filesystem design, each filesystem create or write operation requires two flash blocks to be updated: first the data block and then the metadata block. Buffering is avoided as much as possible to reduce RAM requirements.

However, if the ITS_FLASH_PROGRAM_UNIT is 512 Bytes then the data will have to stored in a temporary memory location in order to be able to write that much data in one-shot.

Proposed implementation overview

1. A new block-sized static buffer should be added to its_flash.c when `ITS_FLASH_PROGRAM_UNIT` is larger than currently supported.
2. Methods calling the flash API such as `its_flash_write()` or `its_flash_block_to_block_move()` will populate the buffer instead of directly programming the flash.
3. A new method `its_flash_flush()`, should be provided in order to flush the block buffer to the device.
4. `its_flash_flush()` should be called twice: Once after a data block update and once more after the metadata block update is completed.
5. The proposed design should require that the data block update is always completed before the metadata block update starts
6. Writes to the block buffer should be atomic, and guarded against corruption by data from different blocks.

Considerations

- The proposed implementation will increase the RAM usage of ITS by the size of a block, only for platforms which require block-aligned writes.
- Currently power-failure is detected by software by incrementing an 8-bit metadata header field (`swap_count`), as the last written byte. When the proposed block-buffer is used, the block is programmed in one-shot and the order the bytes are written on the physical device, is hardware dependent.
- A set of guarantees are required by the supported flash ECC devices. The device's flash APIs should provide a mechanism to capture and raise incomplete program operations, as well as write bytes in a sequential order.

For example, if a board powers down through a 512 page program operation, the next read operation should return an error rather than read invalid data.

Functional flow diagram

The logic of the proposal is described in the following diagram

```
    |----------------------|
    |   data write()       |
    |----------------------|
    |                      |        |------------------------------|
|-> |  its_flash_write  |     --->  | data[] -> its_block_buffer[] |
|   |                      |        |------------------------------|
|   |----------------------|
|   |                      |        |------------------------------------|
|   |   its_flash_flush    |  --->  | its_block_buffer[] -> flash dev IO |
|   |                      |        |------------------------------------|
|   |----------------------|
|             |
|             ------------------------------------
|                                   |
|                                   V
|   |----------------------|        |--------------------------|
|   | data write() complete|        | metadata write() complete|
|   |----------------------|        |--------------------------|
| <-|  Metadata write()    |                    |
    |----------------------|                    |
                                                V
                                    |--------------------------|
                                    |    Operation Complete    |
                                    |--------------------------|
```

PSA Internal Trusted Storage

PSA Internal Trusted Storage (ITS) is a PSA RoT Service for storing the most security-critical device data (e.g. cryptographic keys) in internal storage, which is trusted to provide data confidentiality and authenticity. This contrasts with PSA Protected Storage, which is an Application RoT service that allows larger data sets to be stored securely in external flash, with the option for encryption, authentication and rollback protection to protect the data-at-rest.

Current TF-M Secure Storage

Currently, the TF-M Secure Storage service implements PSA Protected Storage version 1.0-beta2. There is not yet an implementation of PSA Internal Trusted Storage in TF-M.

New TF-M service

The proposal is to implement the *PSA Internal Trusted Storage API* with the *TF-M Internal Trusted Storage service*. It can be abbreviated to *TF-M ITS service* in general and to `its` in code. This name has the advantage of making clear the correspondence between the service and the API it implements.

If this name is adopted, then it may make sense to rename the *Secure Storage service* to the *Protected Storage service* in the future to match. Then "secure storage" could refer to the two services as a collective.

The TF-M ITS service will implement PSA ITS version 1.0. It will be provided by a separate partition to Protected Storage, for a couple of reasons:

- To permit isolation between the services.
  - ITS is a PSA RoT Service, while Protected Storage is an Application RoT Service.
- To avoid circular dependencies.
  - The PSA Firmware Framework does not permit circular dependencies between partitions, which would occur if Protected Storage and ITS were provided by the same partition. Protected Storage depends on Crypto, which in turn depends on ITS.

The existing SST filesystem will be reused to provide the backend of the service, with the flash layer modified to direct storage to internal flash, rather than external.

Compared to Protected Storage, encryption, authentication and rollback protection are not required, so the SST encrypted object layer and the crypto and NV counter interfaces are not required. The rollback protection feature of the object table is also not required.

Code structure

The code structure of the service will be as follows:

TF-M repo:

`interface/`

- `include/psa/internal_trusted_storage.h` - PSA ITS API
- `src/tfm_its_api.c` - PSA ITS API implementation for NSPE

`secure_fw/ns_callable/tfm_veneers.c` - ITS veneers (auto-generated from manifest)

`secure_fw/partitions/internal_trusted_storage/`

- `tfm_internal_trusted_storage.yaml` - Partition manifest
- `tfm_its_secure_api.c` - PSA ITS API implementation for SPE
- `tfm_its_req_mngr.c` - Uniform secure functions and IPC request handlers
- `tfm_internal_trusted_storage.h` - TF-M ITS API (with client_id parameter)
- `tfm_internal_trusted_storage.c` - TF-M ITS implementation, using the flash_fs as a backend
- `its_crypto_interface.h` - APIs for encrypting ITS assets used by ITS implementation (optional)
- `its_crypto_interface.c` - Implementation for ITS encryption (optional)
- `platform/ext/target/.../tfm_hal_its_encryption.c` - Platform implementation for ITS encryption HAL APIs (optional)
- `flash_fs/` - Filesystem
- `flash/` - Flash interface

tf-m-tests repo:

`test/secure_fw/suites/its/`

- `non_secure/psa_its_ns_interface_testsuite.c` - Non-secure interface tests
- `secure/psa_its_s_interface_testsuite.c` - Secure interface tests

TF-M ITS implementation

The following APIs will be exposed by `tfm_internal_trusted_storage.h`:

```
psa_status_t tfm_its_init(void);

psa_status_t tfm_its_set(int32_t client_id,
                         psa_storage_uid_t uid,
                         size_t data_length,
                         const void *p_data,
                         psa_storage_create_flags_t create_flags);

psa_status_t tfm_its_get(int32_t client_id,
                         psa_storage_uid_t uid,
                         size_t data_offset,
                         size_t data_size,
                         void *p_data,
                         size_t *p_data_length);

psa_status_t tfm_its_get_info(int32_t client_id,
                              psa_storage_uid_t uid,
                              struct psa_storage_info_t *p_info);

psa_status_t tfm_its_remove(int32_t client_id,
                            psa_storage_uid_t uid);
```

That is, the TF-M ITS APIs will have the same prototypes as the PSA ITS APIs, but with the addition of a `client_id` parameter, which will be passed from the ITS request manager. A `tfm_its_init` function will also be present, which will be called at initialisation time and not exposed through a veneer or SID.

The implementation in `tfm_internal_trusted_storage.c` must validate the parameters (excepting memory references, which are validated by the SPM), translate the UID and client ID into a file ID and then make appropriate calls to the filesystem layer. It must also take care ensure that any PSA Storage flags associated with the UID are honoured.

Filesystem

The ITS filesystem will be copied and modified from the SST filesystem. The modifications required will be to rename symbols from `sst` to `its` and to update the implementation to be aligned with the latest version of the PSA Storage spec (which consists mainly of moving to the `psa_status_t` error type and using common error codes from `psa/error.h`).

The filesystem will also be modified to align the size of each file stored to the alignment requirement exposed by the flash interface, by adding appropriate padding.

The filesystem code will be de-duplicated again once the ITS service is implemented (see below).

Flash layer

The flash layer will be copied from SST, and modified to direct writes to the internal flash device. It too needs to be updated to use `psa_status_t` error types.

Platform layer

The TF-M platform layer must be be updated to distinguish between the external flash device used for Protected Storage and internal flash device used for ITS. A flash region for the relevant storage service needs to be allocated in each.

On test platforms these may just be two distinct regions of the same flash device, but in general they will separate devices with their own drivers.

Detailed design considerations

Mapping UID onto file ID

The ITS APIs identify assets with 64-bit UIDs, to which the ITS service must append the 32-bit client ID of the calling partition for access control. The existing filesystem uses 32-bit file IDs to identify files, so some mapping would be required to convert between the identifiers.

SST uses the object table to do the mapping from client ID, UID pairs to file IDs, which means making an extra filesystem read/write for each get/set operation. This mapping has minimal overhead for SST though, because object table lookups are already required for rollback protection.

For ITS, no rollback protection feature is required, so there are two options:

- Keep a simplified version of the SST object table that just maps from (client ID, UID) to file ID
- Modify the filesystem to take (at least) 96-bit file IDs, in the form of a fixed-length char buffer.

The advantage of the former is that it would require no extra modification to the existing filesystem code, and the existing SST object table could be cut down for ITS. However, it would mean that every ITS request would invoke twice the number of filesystem operations, increasing latency and flash wear. The code size of the ITS partition would be increased, as would RAM usage as the table would need to be read into RAM.

The latter option would make the filesystem slightly more complex: the size of a metadata entry would be increased by 64-bits and the 96-bit fids would need to be copied and compared with `memcpy` and `memcmp` calls. On the other hand, mapping onto file IDs would incur only the cost of copying the UID and client ID values into the file ID buffer.

A third, even more general, solution would be to use arbitrary-length null-terminated strings as the file IDs. This is the standard solution in full-featured filesystems, but we do not currently require this level of complexity in secure storage.

With this in mind, the proposed option is the second.

Storing create flags

The ITS APIs provide a 32-bit `create_flags` parameter, which contains bit flags that determine the properties of the stored data. Only one flag is currently defined for ITS: `PSA_STORAGE_FLAG_WRITE_ONCE`, which prevents a UID from being modified or deleted after it is set for the first time.

There are two places that these flags could be stored: in the file data or as part of the file metadata.

For the first option, the ITS implementation would need to copy to the flags into the buffer containing the data, and adjust the size accordingly, for each set operation, and the reverse for each get. Every get_info operation would need to read some of the file data, rather than just the metadata, implying a second flash read. A potential downside is that many of the cryptographic assets stored in ITS will be aligned to power-of-two sizes; adding an extra 32-bits would misalign the size, which may reduce flash performance or necessitate adding padding to align to the flash page size.

To implement the second option, a 32-bit `flag` field would be added to the filesystem's metadata structure, whose interpretation is defined by the user. This field would clearly be catered towards the PSA Storage APIs, even if nominally generic, and alternative filesystems may not have any such field. However, it is a more intuitive solution and would simplify both flash alignment and get_info operations.

Overall, it seems more beneficial to store the flags in the metadata, so this is the proposed solution.

Code sharing between Protected Storage and ITS

To de-duplicate the filesystem code used by both Protected Storage and ITS, it is proposed that Protected Storage calls ITS APIs as its backend filesystem.

Protected Storage essentially becomes an encryption, authentication and rollback protection layer on top of ITS. It makes IPC requests or secure function calls to the ITS service to do filesystem operations on its behalf.

This has a couple of advantages:

- It shrinks Protected Storage's stack size, because the filesystem and flash layer stack is only in ITS.
- It automatically solves the problem of ensuring mutual exclusion in the filesystem and flash layers when Protected Storage and ITS are called concurrently. The second request to ITS will just be made to wait by the SPM.

The disadvantage of this approach is that it will increase the latency of Protected Storage requests, due to the extra overhead associated with making a second IPC request or secure function call. It also limits Protected Storage to using only the ITS APIs, unless extra veneers are added solely for Protected Storage to use. This, for example, prevents Protected Storage from doing partial writes to file without reading and re-writing the whole file.

ITS will need to be modified to direct calls from Protected Storage to a different flash device. It can use the client ID to detect when the caller is Protected Storage, and pass down the identity of the flash device to use to the flash layer, which then calls the appropriate driver.

An open question is what to do if Protected Storage itself wants to store something in internal storage in the future (e.g. rollback counters, hash tree/table or top hash). A couple of possible solutions would be:

- Divide up the UIDs, so certain UIDs from Protected Storage refer to assets in internal storage, and others to ones in external storage.
- Use the `type` field of `psa_call` in IPC model to distinguish between internal and external storage requests.

The other option for code sharing would be for Protected Storage and ITS to directly share filesystem code, which would be placed in a shared code region. With this approach, mutual exclusion to the flash device would need to be implemented separately, as would some way of isolating static memory belonging to each partition but not the code. Because of these complications, this option has not been considered further at this time.

Encryption in ITS

The ITS can optionally be configured to encrypt the internal trusted storage data. To support encryption in ITS the target platform must provide an implementation of the APIs defined in `platform/include/tfm_hal_its_encryption.h`:

```
enum tfm_hal_status_t tfm_hal_its_aead_generate_nonce(uint8_t *nonce,
                                                      const size_t nonce_size);

enum tfm_hal_status_t tfm_hal_its_aead_encrypt(
                                     struct tfm_hal_its_auth_crypt_ctx *ctx,
                                     const uint8_t *plaintext,
```

(continues on next page)

(continued from previous page)

```
                                     const size_t plaintext_size,
                                     uint8_t *ciphertext,
                                     const size_t ciphertext_size,
                                     uint8_t *tag,
                                     const size_t tag_size);

enum tfm_hal_status_t tfm_hal_its_aead_decrypt(
                                     struct tfm_hal_its_auth_crypt_ctx *ctx,
                                     const uint8_t *ciphertext,
                                     const size_t ciphertext_size,
                                     uint8_t *tag,
                                     const size_t tag_size,
                                     uint8_t *plaintext,
                                     const size_t plaintext_size);
```

Then encryption can be enabled by setting the build option `-DITS_ENCRYPTION=ON`.

The figure Figure 18: describes the encryption and decryption process happening when calling `tfm_its_set` and `tfm_its_get`.

Figure 18:: En/Decryption of ITS

By using an AEAD scheme, it is possible to not only encrypt the file data but also authenticate the file meta data, which include:

- File id
- File size
- File flags

The key used to perform the AEAD operation must be derived from a long-term key-derivation key and the file id, which is used as a derivation label. The long-term key-derivation key must be managed by the target platform.

11.3.9 Firmware Update Service

Author
Sherry Zhang

Organization
Arm Limited

Contact
Sherry Zhang <Sherry.Zhang2@arm.com>

Table of Contents

Introduction of Firmware Update service

The Firmware Update(FWU) service provides the functionality of updating firmware images. It provides a standard interface for updating firmware and it is platform independent. TF-M defines a shim layer to support cooperation between bootloader and FWU service.

This partition supports the following features:

- Query the firmware store information.
- Image preparation: prepare a new firmware image in the component's firmware store.
- Image installation: install prepared firmware images on all components that have been prepared for installation.
- Image trial: manage a trial of new firmware images atomically on all components that are in TRIAL state.

A typical flow through the component states is shown below[1].

Components

The structure of the TF-M Firmware Update service is listed below:

[1] PSA Firwmare Update API

Component name	Description	Location
Client API interface	This module exports the client API of PSA Firmware Update to the users.	`./interface/src/tfm_fwu_api.c`
Manifest	The manifest file is a description of the service components.	`./secure_fw/partitions/firmware_update/tfm_firmware_update.yaml`
NSPE client API interface	This module exports the client API of PSA Firmware Update to the NSPE(i.e. to the applications).	`./interface/src/tfm_fwu_api.c`
IPC request handlers	This module handles all the secure requests in IPC model. It maitains the image state context and calls the image ID converter to achieve the firmware update functionalities.	`./secure_fw/partitions/firmware_update/tfm_fwu_req_mngr.c`
Shim layer between FWU and bootloader	This module provides the APIs with the functionality of operating the bootloader to cooperate with the Firmware Update service	`./secure_fw/partitions/firmware_update/bootloader/tfm_bootloader_fwu_abstraction.h`
Shim layer example based on MCUboot	This module is the implementation of the shim layer between FWU and bootloader based on MCUboot.	`./secure_fw/partitions/firmware_update/bootloader/mcuboot/tfm_mcuboot_fwu.c`

Service API description

This service follows the PSA Firmware Update API spec of version 1.0[Page 397, 1]. Please refer to Firmware Update spec for the detailed description.

Shim Layer between FWU and bootloader

The firmware update operations are achieved by calling the shim layer APIs between bootloader and FWU.

Shim layer introduction

This shim layer provides the APIs with the functionality of operating the bootloader to cooperate with the Firmware Update service. This shim layer is decoupled from bootloader implementation. Users can specify a specific bootloader by setting TFM_FWU_BOOTLOADER_LIB build configuration and adding the specific build scripts into that file. By default, the MCUboot is chosen as the bootloader.

Interfaces of the shim Layer

fwu_bootloader_init(function)

Prototype

```
psa_status_t fwu_bootloader_init(void);
```

Description

Bootloader related initialization for the firmware update. It reads some necessary shared data from the memory if needed. It initializes the flash drivers defined in FLASH_DRIVER_LIST. Platform can define FLASH_DRIVER_LIST in flash_layout.h to overload the default driver list.

Parameters

N/A

fwu_bootloader_staging_area_init(function)

Prototype

```
psa_status_t fwu_bootloader_staging_area_init(psa_fwu_component_t component,
                                              const void *manifest,
                                              size_t manifest_size);
```

Description

The component is in READY state. Prepare the staging area of the component for image download. For example, initialize the staging area, open the flash area, and so on.

Parameters

- component: The identifier of the target component in bootloader.
- manifest: A pointer to a buffer containing a detached manifest for the update. If the manifest is bundled with the firmware image, manifest must be NULL.
- manifest_size: Size of the manifest buffer in bytes.

fwu_bootloader_load_image(function)

Prototype

```
psa_status_t fwu_bootloader_load_image(psa_fwu_component_t component,
                                       size_t              image_offset,
                                       const void          *block,
                                       size_t              block_size);
```

Description

Load the image into the target component.

Parameters

- `component`: The identifier of the target component in bootloader.
- `image_offset`: The offset of the image being passed into block, in bytes.
- `block`: A buffer containing a block of image data. This might be a complete image or a subset.
- `block_size`: Size of block.

fwu_bootloader_install_image(function)

Prototype

```
psa_status_t fwu_bootloader_install_image(psa_fwu_component_t *candidates,
                                          uint8_t number);
```

Description

Check the authenticity and integrity of the image. If a reboot is required to complete the check, then mark this image as a candidate so that the next time bootloader runs it will take this image as a candidate one to bootup. Return the error code PSA_SUCCESS_REBOOT.

Parameters

- `candidates`: A list of components in CANDIDATE state.
- `number`: Number of components in CANDIDATE state.

fwu_bootloader_mark_image_accepted(function)

Prototype

```
psa_status_t fwu_bootloader_mark_image_accepted(const psa_fwu_component_t *trials,
                                                uint8_t number);
```

Description

Call this API to mark the TRIAL(running) image in component as confirmed to avoid revert when next time bootup. Usually, this API is called after the running images have been verified as valid.

Parameters

- `trials`: A list of components in TRIAL state.
- `number`: Number of components in TRIAL state.

fwu_bootloader_reject_staged_image(function)

Prototype

```
psa_status_t fwu_bootloader_reject_staged_image(psa_fwu_component_t component);
```

Description

The component is in STAGED state. Call this API to Uninstall the staged image in the component so that this image will not be treated as a candidate next time bootup.

Parameters

- component: The identifier of the target component in bootloader.

fwu_bootloader_reject_trial_image(function)

Prototype

```
psa_status_t fwu_bootloader_reject_trial_image(psa_fwu_component_t component);
```

Description

The component is in TRIAL state. Mark the running image in the component as rejected.

Parameters

- component: The identifier of the target component in bootloader.

fwu_bootloader_clean_component(function)

Prototype

```
psa_status_t fwu_bootloader_clean_component(psa_fwu_component_t component);
```

Description

The component is in FAILED or UPDATED state. Clean the staging area of the component.

Parameters

- component: The identifier of the target component in bootloader.

fwu_bootloader_get_image_info(function)

Prototype

```
psa_status_t fwu_bootloader_get_image_info(psa_fwu_component_t component,
                                           bool query_state,
                                           bool query_impl_info,
                                           psa_fwu_component_info_t *info);
```

Description

Get the image information of the given bootloader_image_id in the staging area or the running area.

Parameters

- `component`: The identifier of the target component in bootloader.
- `query_state`: Whether query the 'state' field of psa_fwu_component_info_t.
- `query_impl_info`: Whether Query 'impl' field of psa_fwu_component_info_t.
- `info`: Buffer containing return the component information.

Additional shared data between BL2 and SPE

An additional TLV area "image version" is added into the shared memory between BL2 and TF-M. So that the firmware update partition can get the image version. Even though the image version information is also included in the `BOOT RECORD` TLV area which is encoded by CBOR, adding a dedicated `image version` TLV area is preferred to avoid involving the CBOR encoder which can increase the code size. The FWU partition will read the shared data at the partition initialization.

Build configurations related to FWU partition

- `TFM_PARTITION_FIRMWARE_UPDATE` Controls whether FWU partition is enabled or not.
- `TFM_FWU_BOOTLOADER_LIB` Bootloader configure file for FWU partition.
- `TFM_CONFIG_FWU_MAX_WRITE_SIZE` The maximum permitted size for block in psa_fwu_write, in bytes.
- `TFM_FWU_BUF_SIZE` Size of the FWU internal data transfer buffer (defaults to TFM_CONFIG_FWU_MAX_WRITE_SIZE if not set).
- `FWU_STACK_SIZE` The stack size of FWU Partition.
- `FWU_DEVICE_CONFIG_FILE` The device configuration file for FWU partition. The default value is the configuration file generated for MCUboot. The following macros should be defined in the configuration file:
  - `FWU_COMPONENT_NUMBER` The number of components on the device.

    Note: In this design, component ID ranges from 0 to `FWU_COMPONENT_NUMBER` - 1.

  - `FWU_SUPPORT_TRIAL_STATE` Whether TRIAL component state is supported.
- `TEST_NS_FWU` FWU nonsecure tests switch.
- `TEST_S_FWU` FWU secure tests switch.

> **Note:** The running image which supports revert mechanism should be confirmed before initiating a firmware update process. For example, if the running image is built with `-DMCUBOOT_UPGRADE_STRATEGY=SWAP_USING_MOVE`, the image should be confirmed either by adding `-DMCUBOOT_CONFIRM_IMAGE=ON` build option or by calling `psa_fwu_accept()` API before initiating a firmware update process. Otherwise, `PSA_ERROR_BAD_STATE` will be returned by `psa_fwu_start()`.

Limitations of current implementation

Currently, the MCUboot based implementation does not record image update results like failure or success. And FWU partition does not detect failure errors in bootloader installation. If an image installation fails in the bootloader and the old image still runs after reboot, `PSA_FWU_READY` state will be returned by `psa_fwu_query()` after reboot.

Currently, image download recovery after a reboot is not supported. If a reboot happens in image preparation, the downloaded image data will be ignored after the reboot.

Benefits Analysis on this Partition

Implement the FWU functionality in the non-secure side

The APIs listed in PSA Firmware Update API spec[Page 397, 1] can also be implemented in the non-secure side.

Pros and Cons for implementing FWU APIs in secure side

Pros

- It protects the image in the passive or staging area from being tampered with by the NSPE. Otherwise, a malicious actor from NSPE can tamper the image stored in the non-secure area to break image update.
- It protects secure image information from disclosure. In some cases, the non-secure side shall not be permitted to get secure image information.
- It protects the active image from being manipulated by NSPE. Some bootloader supports testing the image. After the image is successfully installed and starts to run, the user should set the image as permanent image if the image passes the test. To achieve this, the area of the active image needs to be accessed. In this case, implementing FWU service in SPE can prevent NSPE from manipulating the active image area.
- On some devices, such as the Arm Musca-B1 board, the passive or staging area is restricted as secure access only. In this case, the FWU partition should be implemented in the secure side.

Cons

- It increases the image size of the secure image.
- It increases the execution latency and footprint. Compared to implementing FWU in NSPE directly, calling the Firmware Update APIs which are implemented in the secure side increases the execution latency and footprint.
- It can increase the attack surface of the secure runtime.

Users can decide whether to call the FWU service in TF-M directly or implement the Firmware Update APIs in the non-secure side based on the pros and cons analysis above.

Reference

11.3.10 Protected Storage service key management

Author
Jamie Fox

Organization
Arm Limited

Contact
Jamie Fox <jamie.fox@arm.com>

Background

The PSA Protected Storage API requires confidentiality for external storage to be provided by:

> **cryptographic ciphers using device-bound keys**, a tamper resistant enclosure, or an inaccessible deployment location, depending on the threat model of the deployed system.

A TBSA-M-compliant device must embed a Hardware Unique Key (HUK), which provides the root of trust (RoT) for confidentiality in the system. It must have at least 128 bits of entropy (and a 128 bit data size), and be accessible only to Trusted code or Trusted hardware that acts on behalf of Trusted code. [TBSA-M]

Design description

Each time the system boots, PS will request that the Crypto service uses a key derivation function (KDF) to derive a storage key from the HUK, by referring to the builtin key handle for the HUK. The storage key could be kept in on-chip volatile memory private to the Crypto partition, or it could remain inside a secure element. Either way it will not be returned to PS.

For each call to the PSA Protected Storage APIs, PS will make requests to the Crypto service to perform AEAD encryption and/or decryption operations using the storage key (providing a fresh nonce for each encryption).

At no point will PS access the key material itself, only referring to the HUK and storage key by their handles in the Crypto service.

Key derivation

PS will make key derivation requests to the Crypto service with calls to the PSA Crypto APIs. In order to derive the storage key, the following calls are required:

```
status = psa_key_derivation_setup(&op, PSA_ALG_HKDF(PSA_ALG_SHA_256));

/* Set up a key derivation operation with HUK  */
status = psa_key_derivation_input_key(&op, PSA_KEY_DERIVATION_INPUT_SECRET,
                                      TFM_BUILTIN_KEY_ID_HUK);

/* Supply the PS key label as an input to the key derivation */
```

(continues on next page)

(continued from previous page)

```
status = psa_key_derivation_input_bytes(&op, PSA_KEY_DERIVATION_INPUT_INFO,
                                        key_label,
                                        key_label_len);

/* Create the storage key from the key derivation operation */
status = psa_key_derivation_output_key(&attributes, &op, &ps_key);
```

Note: TFM_BUILTIN_KEY_ID_HUK is a static key ID that is used to identify the HUK. It has an arbitrary value defined in tfm_crypto_defs.h

ps_key is a PSA Crypto key handle to a volatile key, set by the derivation operation. After the call to psa_key_derivation_output_key, it can be used to refer the storage key.

key_label can be any string that is independent of the input key material and different to the label used in any other derivation from the same input key. It prevents two different contexts from deriving the same output key from the same input key.

The key derivation function used by the crypto service to derive the storage key will be HKDF, with SHA-256 as the underlying hash function. HKDF is suitable because:

- It is simple and efficient, requiring only two HMAC operations when the length of the output key material is less than or equal to the hash length (as is the case here).
- The trade-off is that HKDF is only suitable when the input key material has at least as much entropy as required for the output key material. But this is the case here, as the HUK has 128 bits of entropy, the same as required by PS.
- HKDF is standardised in RFC 5869 [RFC5869] and its security has been formally analysed. [HKDF]
- It is supported by the TF-M Crypto service.

The choice of underlying hash function is fairly straightforward: it needs to be a modern standardised algorithm, considered to be secure and supported by TF-M Crypto. This narrows it down to just the SHA-2 family. Of the hash functions in the family, SHA-256 is the simplest and provides more than enough output length.

Keeping the storage key private to PS

The Crypto service derives a platform key from the HUK, using the partition ID as the input to that derivation, and that platform key is used for the key derivation by PS. This happens transparently, and to PS is indistinguishable from deriving from the HUK except that other partitions cannot derive the storage key even if they know the derivation parameters.

Key use

To encrypt and decrypt data, PS will call the PSA Crypto AEAD APIs in the same way as the current implementation, but ps_key will refer to the storage key, rather than the imported HUK. For each encryption operation, the following call is made (and analogously for decryption):

```
psa_aead_encrypt(ps_key, PS_CRYPTO_ALG,
                 crypto->ref.iv, PS_IV_LEN_BYTES,
                 add, add_len,
                 in, in_len,
                 out, out_size, out_len);
```

References

11.4 Software Design Notes

11.4.1 Code sharing between independently linked XIP binaries

Author
Tamas Ban

Organization
Arm Limited

Contact
tamas.ban@arm.com

Motivation

Cortex-M devices are usually constrained in terms of flash and RAM. Therefore, it is often challenging to fit bigger projects in the available memory. The PSA specifications require a device to both have a secure boot process in place at device boot-up time, and to have a partition in the SPE which provides cryptographic services at runtime. These two entities have some overlapping functionality. Some cryptographic primitives (e.g. hash calculation and digital signature verification) are required both in the bootloader and the runtime environment. In the current TF-M code base, both firmware components use the mbed-crypto library to implement these requirements. During the build process, the mbed-crpyto library is built twice, with different configurations (the bootloader requires less functionality) and then linked to the corresponding firmware component. As a result of this workflow, the same code is placed in the flash twice. For example, the code for the SHA-256 algorithm is included in MCUboot, but the exact same code is duplicated in the SPE cryptography partition. In most cases, there is no memory isolation between the bootloader and the SPE, because both are part of the PRoT code and run in the secure domain. So, in theory, the code of the common cryptographic algorithms could be reused among these firmware components. This could result in a big reduction in code footprint, because the cryptographic algorithms are usually flash hungry. Code size reduction can be a good opportunity for very constrained devices, which might need to use TF-M Profile Small anyway.

Technical challenge

Code sharing in a regular OS environment is easily achievable with dynamically linked libraries. However, this is not the case in Cortex-M systems where applications might run bare-metal, or on top of an RTOS, which usually lacks dynamic loading functionality. One major challenge to be solved in the Cortex-M space is how to share code between independently linked XIP applications that are tied to a certain memory address range to be executable and have absolute function and global data memory addresses. In this case, the code is not relocatable, and in most cases, there is no loader functionality in the system that can perform code relocation. Also, the lack of an MMU makes the address space flat, constant and not reconfigurable at runtime by privileged code.

One other difficulty is that the bootloader and the runtime use the same RAM area during execution. The runtime firmware is executed strictly after the bootloader, so normally, it can reuse the whole secure RAM area, as it would be the exclusive user. No attention needs to be paid as to where global data is placed by the linker. The bootloader does not need to retain its state. The low level startup of the runtime firmware can freely overwrite the RAM with its data without corrupting bootloader functionality. However, with code sharing between bootloader and runtime firmware, these statements are no longer true. Global variables used by the shared code must either retain their value or must be reinitialised during low level startup of the runtime firmware. The startup code is not allowed to overwrite the shared global variables with arbitrary data. The following design proposal provides a solution to these challenges.

Design concept

The bootloader is sometimes implemented as ROM code (BL1) or stored in a region of the flash which is lockable, to prevent tampering. In a secure system, the bootloader is immutable code and thus implements a part of the Root of Trust anchor in the device, which is trusted implicitly. The shared code is primarily part of the bootloader, and is reused by the runtime SPE firmware at a later stage. Not all of the bootloader code is reused by the runtime SPE, only some cryptographic functions.

Simplified steps of building with code sharing enabled:

- Complete the bootloader build process to have a final image that contains the absolute addresses of the shared functions, and the global variables used by these functions.
- Extract the addresses of the functions and related global variables that are intended to be shared from the bootloader executable.
- When building runtime firmware, provide the absolute addresses of the shared symbols to the linker, so that it can pick them up, instead of instantiating them again.

The execution flow looks like this:

```
SPE        MCUboot func1()    MCUboot func2()     MCUboot func3()
 |
 |     Hash()
 |------------->|
                |----------------->|
                                   |
                      Return       |
      Return    |<-----------------|
 |<-------------|
 |
 |
 |------------------------------------------------------>|
                                                        |
         Function pointer in shared global data()       |
 |<------------------------------------------------------|
 |
 |                       Return
 |------------------------------------------------------>|
                                                        |
                         Return                         |
 |<------------------------------------------------------|
 |
 |
```

The execution flow usually returns from a shared function back to the SPE with an ordinary function return. So usually, once a shared function is called in the call path, all further functions in the call chain will be shared as well. However, this is not always the case, as it is possible for a shared function to call a non-shared function in SPE code through a global function pointer.

For shared global variables, a dedicated data section must be allocated in the linker configuration file. This area must have the same memory address in both MCUboot's and the SPE's linker files, to ensure the integrity of the variables. For simplicity's sake, this section is placed at the very beginning of the RAM area. Also, the RAM wiping functionality at the end of the secure boot flow (that is intended to remove any possible secrets from the RAM) must not clear this area. Furthermore, it must be ensured that the linker places shared globals into this data section. There are two way to achieve this:

- Put a filter pattern in the section body that matches the shared global variables.
- Mark the global variables in the source code with special attribute *__attribute__((section(<NAME_OF_SHARED_SYMBOL_SECTION>)))*

RAM memory layout in MCUboot with code sharing enabled:

```
+------------------+
|  Shared symbols  |
+------------------+
| Shared boot data |
+------------------+
|      Data        |
+------------------+
|    Stack (MSP)   |
+------------------+
|      Heap        |
+------------------+
```

RAM memory layout in SPE with code sharing enabled:

```
+-------------------+
|  Shared symbols   |
+-------------------+
| Shared boot data  |
+-------------------+
|    Stack (MSP)    |
+-------------------+
|    Stack (PSP)    |
+-------------------+
| Partition X Data  |
+-------------------+
| Partition X Stack |
+-------------------+
          .
          .
          .
+-------------------+
| Partition Z Data  |
+-------------------+
| Partition Z Stack |
+-------------------+
|     PRoT Data     |
+-------------------+
|       Heap        |
+-------------------+
```

Patching Mbed TLS

In order to share some global function pointers from mbed-crypto that are related to dynamic memory allocation, their scope must be extended from private to global. This is needed because some compiler toolchain only extract the addresses of public functions and global variables, and extraction of addresses is a requirement to share them among binaries. Therefore, a short patch was created for the mbed-crypto library, which "globalises" these function pointers:

lib/ext/mbedcrypto/0002-Enable-crypto-code-sharing-between-independent-binar.patch

The patch needs to be manually applied in the Mbed TLS repo, if code sharing is enabled. The patch has no effect on the functional behaviour of the cryptographic library, it only extends the scope of some variables.

Tools support

All the currently supported compilers provide a way to achieve the above objectives. However, there is no standard way, which means that the code sharing functionality must be implemented on a per compiler basis. The following steps are needed:

- Extraction of the addresses of all global symbols.
- The filtering out of the addresses of symbols that aren't shared. The goal is to not need to list all the shared symbols by name. Only a simple pattern has to be provided, which matches the beginning of the symbol's name. Matching symbols will be shared. Examples are in : *bl2/shared_symbol_template.txt*
- Provision of the addresses of shared symbols to the linker during the SPE build process.
- The resolution of symbol collisions during SPE linking. Because mbed-crypto is linked to both firmware components as a static library, the external shared symbols will conflict with the same symbols found within it. In order to prioritize the external symbol, the symbol with the same name in mbed-crypto must be marked as weak in the symbol table.

The above functionalities are implemented in the toolchain specific CMake files:

- *toolchain_ARMCLANG.cmake*
- *toolchain_GNUARM.cmake*

By the following two functions:

- *target_share_symbols()*: Extract and filter shared symbol addresses from MCUboot.
- *target_link_shared_code()*: Link shared code to the SPE and resolve symbol conflict issues.

ARMCLANG

The toolchain specific steps are:

- Extract all symbols from MCUboot: add *-symdefs* to the compiler command line
- Filter shared symbols: call CMake script *FilterSharedSymbols.cmake*
- Weaken duplicated (shared) symbols in the mbed-crypto static library that are linked to the SPE: *arm-none-eabi-objcopy*
- Link shared code to SPE: Add the filtered output of *-symdefs* to the SPE source file list.

GNUARM

The toolchain specific steps are:

- Extract all symbols from MCUboot: *arm-none-eabi-nm*
- Filter shared symbols: call CMake script: *FilterSharedSymbols.cmake*
- Strip unshared code from MCUboot: *arm-none-eabi-strip*
- Weaken duplicated (shared) symbols in the mbed-crypto static library that are linked to the SPE: *arm-none-eabi-objcopy*
- Link shared code to SPE: Add *-Wl -R <SHARED_STRIPPED_CODE.axf>* to the compiler command line

IAR

Functionality currently not implemented, but the toolchain supports doing it.

Memory footprint reduction

Build type: MinSizeRel Platform: mps2/an521 Version: TF-Mv1.2.0 + code sharing patches MCUboot image encryption support is disabled.

	ConfigDefault		ConfigProfile-M		ConfigProfile-S	
	ARM-CLANG	GNUARM	ARM-CLANG	GNUARM	ARM-CLANG	GNUARM
CODE_SHARING=OFF	122268	124572	75936	75996	50336	50224
CODE_SHARING=ON	113264	115500	70400	70336	48840	48988
Difference	9004	9072	5536	5660	1496	1236

If MCUboot image encryption support is enabled then saving could be up to ~13-15KB.

Note: Code sharing on Musca-B1 was tested only with SW only crypto, so crypto hardware acceleration must be turned off: -DCRYPTO_HW_ACCELERATOR=OFF

Useability considerations

Functions that only use local variables can be shared easily. However, functions that rely on global variables are a bit tricky. They can still be shared, but all global variables must be placed in the shared symbol section, to prevent overwriting and to enable the retention of their values.

Some global variables might need to be reinitialised to their original values by runtime firmware, if they have been used by the bootloader, but need to have their original value when runtime firmware starts to use them. If so, the reinitialising functionality must be implemented explicitly, because the low level startup code in the SPE does not initialise the shared variables, which means they retain their value after MCUboot stops running.

If a bug is discovered in the shared code, it cannot be fixed with a firmware upgrade, if the bootloader code is immutable. If this is the case, disabling code sharing might be a solution, as the new runtime firmware could contain the fixed code instead of relying on the unfixed shared code. However, this would increase code footprint.

API backward compatibility also can be an issue. If the API has changed in newer version of the shared code. Then new code cannot rely on the shared version. The changed code and all the other shared code where it is referenced from must be ignored and the updated version of the functions must be compiled in the SPE binary. The Mbed TLS library is API compatible with its current version (`v2.24.0`) since the `mbedtls-2.7.0 release` (2018-02-03).

To minimise the risk of incompatibility, use the same compiler flags to build both firmware components.

The artifacts of the shared code extraction steps must be preserved so as to remain available if new SPE firmware (that relies on shared code) is built and released. Those files are necessary to know the address of shared symbols when linking the SPE.

How to use code sharing?

Considering the above, code sharing is an optional feature, which is disabled by default. It can be enabled from the command line with a compile time switch:

- *TFM_CODE_SHARING*: Set to *ON* to enable code sharing.

With the default settings, only the common part of the mbed-crypto library is shared, between MCUboot and the SPE. However, there might be other device specific code (e.g. device drivers) that could be shared. The shared cryptography code consists mainly of the SHA-256 algorithm, the *bignum* library and some RSA related functions. If image encryption support is enabled in MCUboot, then AES algorithms can be shared as well.

Sharing code between the SPE and an external project is possible, even if MCUboot isn't used as the bootloader. For example, a custom bootloader can also be built in such a way as to create the necessary artifacts to share some of its code with the SPE. The same artifacts must be created like the case of MCUboot:

- ***shared_symbols_name.txt*: Contains the name of the shared symbols. Used by**
  the script that prevents symbol collision.
- *shared_symbols_address.txt*: Contains the type, address and name of shared symbols. Used by the linker when linking runtime SPE.
- *shared_code.axf*: GNUARM specific. The stripped version of the firmware component, only contains the shared code. It is used by the linker when linking the SPE.

Note: The artifacts of the shared code extraction steps must be preserved to be able to link them to any future SPE version.

When an external project is sharing code with the SPE, the *SHARED_CODE_PATH* compile time switch must be set to the path of the artifacts mentioned above.

Further improvements

This design focuses only on sharing the cryptography code. However, other code could be shared as well. Some possibilities:

- Flash driver
- Serial driver
- Image metadata parsing code
- etc.

11.4.2 Hardware Abstraction Layer

Organization
Arm Limited

Contact
tf-m@lists.trustedfirmware.org

API Version
0.9

Introduction

TF-M HAL abstracts the hardware-oriented and platform specific operations on the *SPE* side and provides a set of APIs to the upper layers such as *SPM*, *RoT Service*. The *HAL* aims to cover the platform different aspects whereas common architecturally defined aspects are done generically within the common *SPE*. In some cases, although the operations are defined architecturally, it may not be possible to generalize implementations because lots of information is only known to platforms. It is more efficient to define a *HAL* API for those architectural operations as well.

Note: *TBSA-M* provides the hardware requirements for security purposes. *TF-M HAL* tries to reference *TBSA-M* recommendations in the interfaces from the software perspective only. Please reference *TBSA-M* for your security hardware design.

Design Goals

TF-M HAL is designed to simplify the integration efforts on different platforms.

TF-M HAL is designed to make it easy to use the hardware and develop the *SPM* and *RoT Service* which need to access the devices.

TF-M HAL is designed to make the structure clearer and let the *TF-M* mainly focus on *PSA* implementation.

Overview

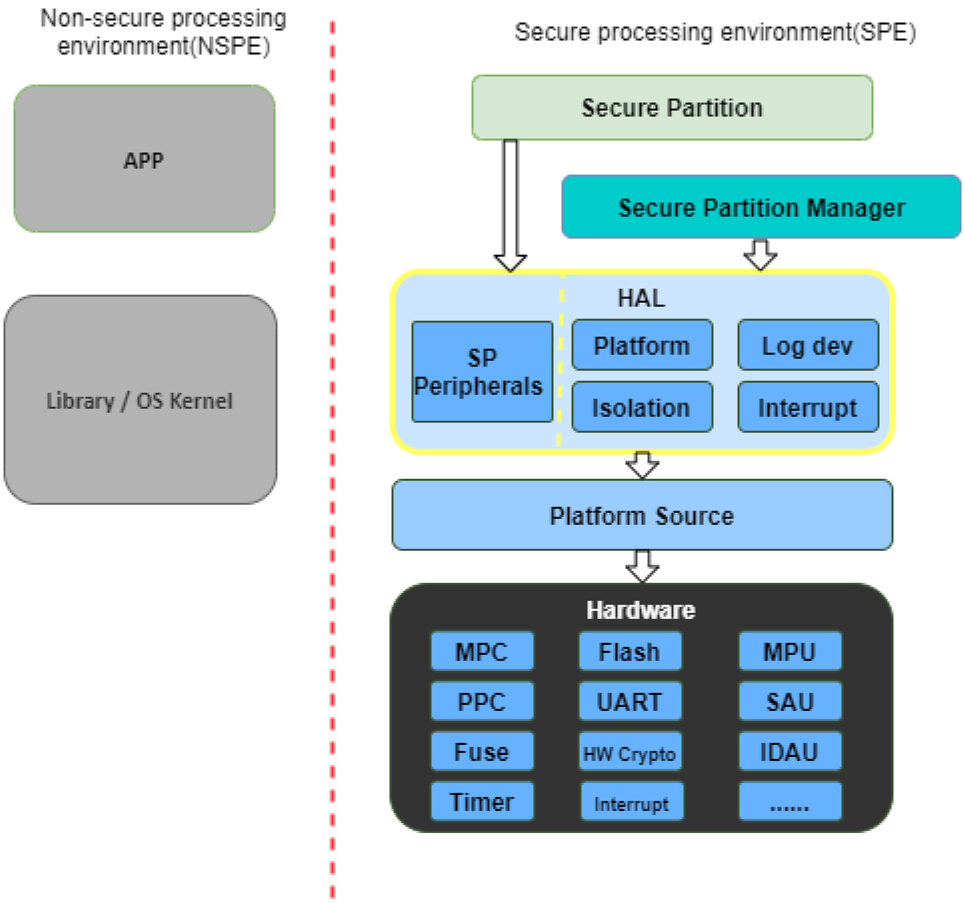
This section provides an overview of the abstraction layer structure.

Here lists a minimal set of necessary functionalities:

- **Isolation API**: Provides the necessary isolation functionalities required by the *PSA-FF-M* and *TBSA-M*, and provides APIs to *SPM* to check the permissions of memory access.
- **Platform API**: Provides the platform initialization, platform-specific memory information, system reset, etc.
- **Log dev API**: Provides the log system functions.
- **Interrupt API**: Provides the interrupt functions.

Note:

- There is a non-secure *HAL* that focuses on the mailbox operation API for Dual-core topology. For more information about it, please refer to *Mailbox Design in TF-M on Dual-core System*.
- The minimal set of *TF-M HAL* is sufficient for Secure Partitions by using customized peripheral interfaces. To provide easier portability for the Secure Partitions, a Secure Partition *HAL* is provided in this design too.


Non-secure processing environment(NSPE)
APP
Library / OS Kernel
Secure processing environment(SPE)
Secure Partition
Secure Partition Manager
HAL
SP Peripherals
Platform
Log dev
Isolation
Interrupt
Platform Source
Hardware
MPC
Flash
MPU
PPC
UART
SAU
Fuse
HW Crypto
IDAU
Timer
Interrupt
......

- The debug mechanisms give the external entity the corresponding right to access the system assets. *TF-M* ensures that the external entity is permitted access to those assets. Currently, *TF-M* only needs the debug authentication. The whole debug mechanism and related *HAL* will be enhanced in the future. Please refer to the *Debug authentication settings section* for more details.

Design Principles

As *TF-M* runs on resource-constrained devices, the *HAL* tries to avoid multiple level abstractions which cost more resources.

Part of the *HAL* interfaces does not focus on exact hardware operations such as power on/off or PIN manipulation. Instead, the *HAL* abstracts higher-level interfaces to reserve the implementation flexibility for the platform vendors.

The *TF-M HAL* should be easy to deprecate APIs and provide compatibilities. Any API incompatibility should be detected during building.

TF-M relies on the *HAL* APIs to be implemented correctly and trusts the *HAL* APIs. *TFM* can provide assertions to detect common programming errors but essentially no further extensive checks will be provided.

Source Files

This section describes the source file of the *TF-M HAL*, including the header and c files.

tfm_hal_defs.h

This header file contains the definitions of common macros and types used by all *HAL* APIs. Please refer to *Status Codes* for detailed definitions.

tfm_hal_[module].[h/c]

All other headers and c files are classified by the modules, such as isolation, platform, interrupt, devices, etc.

Note: There are common files in the platform folder include the implemented *HAL* APIs. The platform vendors can use them directly but need to implement all the sub APIs.

Status Codes

These are common status and error codes for all *HAL* APIs.

Types

tfm_hal_status_t

This is a status code to be used as the return type of *HAL* APIs.

```
enum tfm_hal_status_t {
    TFM_HAL_ERROR_MEM_FAULT = SCHAR_MIN,
    TFM_HAL_ERROR_MAX_VALUE,
    TFM_HAL_ERROR_BAD_STATE,
    TFM_HAL_ERROR_NOT_SUPPORTED,
    TFM_HAL_ERROR_INVALID_INPUT,
    TFM_HAL_ERROR_NOT_INIT,
    TFM_HAL_ERROR_GENERIC,
    TFM_HAL_SUCCESS = 0
};
```

Error Codes

Negative values indicate an error. Zero and positive values indicate success.

Here is the general list. The detailed usages for each error code are described in the API introduction sections.

TFM_HAL_SUCCESS

Status code to indicate general success.

TFM_HAL_ERROR_GENERIC

Status code to indicate an error that does not correspond to any defined failure cause.

TFM_HAL_ERROR_NOT_INIT

Status code to indicate that the module is not initialed.

TFM_HAL_ERROR_INVALID_INPUT

Status code to indicate that the input is invalid.

TFM_HAL_ERROR_NOT_SUPPORTED

Status code to indicate that the requested operation or a parameter is not supported.

TFM_HAL_ERROR_BAD_STATE

Status code to indicate that the requested action cannot be performed in the current state.

TFM_HAL_ERROR_MAX_VALUE

Status code to indicate that the current number has got the max value.

TFM_HAL_ERROR_MEM_FAULT

Status code to indicate that the memory check failed.

API Definition for TF-M SPM

This section describes the APIs defined for *TF-M SPM*.

Platform API

The platform API is a higher-level abstraction layer of the platform, other than a dedicated API set for the special hardware devices.

APIs

tfm_hal_platform_init()

Prototype

```
enum tfm_hal_status_t tfm_hal_platform_init(void)
```

Description

This API performs the platform initializations **before** the *SPM* initialization.

The initializations could include but not limited to: - Fault handlers - Reset configurations - Debug init - NVIC init

Parameter

- `void` - None.

Return Values

- `TFM_HAL_SUCCESS` - Init success.
- `TFM_HAL_ERROR_GENERIC` - Generic errors.

tfm_hal_system_reset()

Prototype

```
void tfm_hal_system_reset(void)
```

Description

This API performs a system reset.

The platform can uninitialize some resources before reset.

When `CONFIG_TFM_HALT_ON_CORE_PANIC` is disabled this function is called to reset the system when a fatal error occurs.

Parameter

- `void` - None

Return Values

- `void` - None

Note

This API should not return.

tfm_hal_system_halt()

Prototype

```
void tfm_hal_system_halt(void)
```

Description

This API enters the CPU into an infinite loop.

The platform can uninitialize some resources before looping forever.

When `CONFIG_TFM_HALT_ON_CORE_PANIC` is enabled this function is called to halt the system when a fatal error occurs.

Parameter

- `void` - None

Return Values

- `void` - None

Note

This API should not return.

Isolation API

The *PSA-FF-M* defines three isolation levels and a memory access rule to provide diverse levels of security. The isolation API provides the functions to implement these requirements.

The Isolation API operates on boundaries. A boundary represents a set of protection settings that isolates components and domains. Below are the boundary examples in the current implementation:

- Boundaries between SPM and Secure Partitions.
- Boundaries between ARoT domain and PRoT domain.

There are two types of boundaries:

- Static boundaries: Set up when the system is initializing and persistent after the initialization. This type of boundary needs the set-up operations only.
- Partition boundaries: Keeps switching from one to another when the system is running. This type of boundary needs both set-up and switching operations.

The boundary operations are abstracted as HAL interfaces because isolation hardwares can be different for platforms:

- The set-up HAL interface creates a partition boundary based on given partition information. This created boundary is bound with the partition for subsequent usage. The binding is done by storing the boundary into partition runtime data.
- The activation HAL interface activates the partition boundary to secure the execution for the partition to be switched. The target partition's information and boundary are given to the activation HAL to accomplish the operation.

The data representing the partition boundary in runtime is defined with the opaque type `uintptr_t`:

- It is required that one value represents one boundary. The different values represent different boundaries.
- The value is created by HAL implementation with its own-defined encoding scheme.

The HAL implementation defined encoding scheme can be designed for implementation convenience. For example:

- The implementation scheme can encode attribute flags into integer bits. This could help the activation HAL to extract the protection settings quickly from this encoded value, or even write to hardware registers directly in the most ideal case. The initial TF-M Isolation HAL reference implementation applies this scheme.
- The implementation scheme can reference the addresses of isolation hardware description data. This could help the activation HAL to reference the protection settings directly by pointers.

Multiple Secure Partitions can bind with the same boundary value. This flexibility is useful for specific configurations. Take Isolation Level 2 as an example, assigning PRoT and ARoT domain boundaries to respective partitions can make execution more efficient, because switching two partitions in the same domain does not need to change the activated boundary.

The boundary contains the partition's memory accessibility information, hence memory access check shall be performed based on boundary.

Memory Access Attributes

The memory access attributes are encoded as bit fields, you can logic OR them to have a combination of the atrributes, for example `TFM_HAL_ACCESS_UNPRIVILEGED | TFM_HAL_ACCESS_READABLE` is unprivileged readable. The data type is *uint32_t*.

TFM_HAL_ACCESS_EXECUTABLE

The memory is executable.

```
#define TFM_HAL_ACCESS_EXECUTABLE (1UL << 0)
```

TFM_HAL_ACCESS_READABLE

The memory is readable.

```
#define TFM_HAL_ACCESS_READABLE (1UL << 1)
```

TFM_HAL_ACCESS_WRITABLE

The memory is writable.

```
#define TFM_HAL_ACCESS_WRITABLE (1UL << 2)
```

TFM_HAL_ACCESS_UNPRIVILEGED

The memory is unprivileged mode accessible.

```
#define TFM_HAL_ACCESS_UNPRIVILEGED (1UL << 3)
```

TFM_HAL_ACCESS_DEVICE

The memory is a MMIO device.

```
#define TFM_HAL_ACCESS_DEVICE (1UL << 4)
```

TFM_HAL_ACCESS_NS

The memory is accessible from *NSPE*

```
#define TFM_HAL_ACCESS_NS (1UL << 5)
```

APIs

tfm_hal_verify_static_boundaries()

Prototype

```
fih_int tfm_hal_verify_static_boundaries(void)
```

Description

This API verifies the static isolation boundaries.

Refer to the *PSA-FF-M* for the definitions of the isolation boundaries.

Parameter

- `void` - None

Return Values

- `TFM_HAL_SUCCESS` - Verification has been successful.
- `TFM_HAL_ERROR_GENERIC` - Verification failed.

tfm_hal_set_up_static_boundaries()

Prototype

```
enum tfm_hal_status_t tfm_hal_set_up_static_boundaries(void)
```

Description

This API sets up the static isolation boundaries which are constant throughout the system runtime.

These boundaries include:

- The boundary between the *SPE* and the *NSPE*
- The boundary to protect the SPM execution. For example, the PSA RoT isolation boundary between the PSA Root of Trust and the Application Root of Trust which is for isolation level 2 and 3 only.

Refer to the *PSA-FF-M* for the definitions of the isolation boundaries.

Parameter

- `void` - None

Return Values

- `TFM_HAL_SUCCESS` - Isolation boundaries have been set up.
- `TFM_HAL_ERROR_GENERIC` - Failed to set up the static boundaries.

tfm_hal_bind_boundary()

Prototype

```
enum tfm_hal_status_t tfm_hal_bind_boundary(
                                const struct partition_load_info_t *p_ldinf,
                                uintptr_t *p_boundary);
```

Description

This API binds partition with a platform-generated boundary. The boundary is bound by writing the generated value into `p_boundary`. And this bound boundary is used in subsequent calls to *tfm_hal_activate_boundary()* when boundary's owner partition get scheduled for running.

Parameter

- `p_ldinf` - Load information of the partition that is under loading.
- `p_boundary` - Pointer for holding a partition's boundary.

Return Values

- `TFM_HAL_SUCCESS` - The boundary has been bound successfully.
- `TFM_HAL_ERROR_GENERIC` - Failed to bind the handle.

tfm_hal_activate_boundary()

Prototype

```
enum tfm_hal_status_t tfm_hal_activate_boundary(
                            const struct partition_load_info_t *p_ldinf,
                            uintptr_t boundary);
```

Description

This API requires the platform to activate the boundary to ensure the given Secure Partition can run successfully.

The access permissions outside the boundary is platform-dependent.

Parameter

- `p_ldinf` - The load information of the partition that is going to be run.
- `boundary` - The boundary for the owner partition of `p_ldinf`. This value is bound in function `tfm_hal_bind_boundary`.

Return Values

- `TFM_HAL_SUCCESS` - the isolation boundary has been set up.
- `TFM_HAL_ERROR_GENERIC` - failed to set up the isolation boundary.

tfm_hal_memory_check()

Prototype

```
tfm_hal_status_t tfm_hal_memory_check(uintptr_t boundary,
                                      uintptr_t base,
                                      size_t size,
                                      uint32_t access_type)
```

Description

This API checks if a given range of memory can be accessed with specified access types in boundary. The boundary belongs to a partition which contains asset info.

Parameter

- `boundary` - Boundary of a Secure Partition. Check *tfm_hal_bind_boundary* for details.
- `base` - The base address of the region.
- `size` - The size of the region.
- `access_type` - The memory access types to be checked between given memory and boundaries. The *Memory Access Attributes*.

Return Values

- `TFM_HAL_SUCCESS` - The memory region has the access permissions.
- `TFM_HAL_ERROR_MEM_FAULT` - The memory region does not have the access permissions.
- `TFM_HAL_ERROR_INVALID_INPUT` - Invalid inputs.
- `TFM_HAL_ERROR_GENERIC` - An error occurred.

Note

If the implementation chooses to encode a pointer as the boundary, a platform-specific pointer validation needs to be considered before referencing the content in this pointer.

tfm_hal_boundary_need_switch()

Prototype

```
bool tfm_hal_boundary_need_switch(uintptr_t boundary_from,
                                  uintptr_t boundary_to)
```

Description

This API let the platform decide if a boundary switch is needed.

Parameter

- `boundary_from` - Boundary to switch from.
- `boundary_to` - Boundary to switch to.

Return Values

- `true` - A switching is needed
- `false` - No need for a boundary switch

Log API

The log API is used by the *TF-M log system*. The log device could be uart, memory, usb, etc.

APIs

tfm_hal_output_partition_log()

Prototype

```
int32_t tfm_hal_output_partition_log(const unsigned char *str, uint32_t len)
```

Description

This API is called by Secure Partition to output logs.

Parameter

- `str` - The string to output.
- `len` - Length of the string in bytes.

Return Values

- Positive values - Number of bytes output.
- `TFM_HAL_ERROR_NOT_INIT` - The log device has not been initialized.
- `TFM_HAL_ERROR_INVALID_INPUT` - Invalid inputs when `str` is `NULL` or `len` is zero.

Note

None.

tfm_hal_output_spm_log()

Prototype

```
int32_t tfm_hal_output_spm_log(const unsigned char *str, uint32_t len)
```

Description

This API is called by *SPM* to output logs.

Parameter

- `str` - The string to output.
- `len` - Length of the string in bytes.

Return Values

- Positive numbers - Number of bytes output.
- `TFM_HAL_ERROR_NOT_INIT` - The log device has not been initialized.
- `TFM_HAL_ERROR_INVALID_INPUT` - Invalid inputs when `str` is `NULL` or `len` is zero.

Note

Please check *TF-M log system* for more information.

Interrupt APIs

The SPM HAL interrupt APIs are intended for operations on Interrupt Controllers of platforms.

APIs for control registers of interrupt sources are not in the scope of this set of APIs. Secure Partitions should define the APIs for managing interrupts with those MMIO registers.

APIs

tfm_hal_irq_enable()

Prototype

```
enum tfm_hal_status_t tfm_hal_irq_enable(uint32_t irq_num)
```

Description

This API enables an interrupt from the Interrupt Controller of the platform.

Parameter

- `irq_num` - the interrupt to be enabled with a number

Return Values

- `TFM_HAL_ERROR_INVALID_INPUT` - the `irq_num` exceeds The maximum supported number of external interrupts.
- `TFM_HAL_ERROR_GENERIC` - failed to enable the interrupt.
- `TFM_HAL_SUCCESS` - the interrupt is successfully enabled.

tfm_hal_irq_disable()

Prototype

```
enum tfm_hal_status_t tfm_hal_irq_disable(uint32_t irq_num)
```

Description

This API disables an interrupt from the Interrupt Controller of the platform.

Parameter

- `irq_num` - the interrupt to be disabled with a number

Return Values

- `TFM_HAL_ERROR_INVALID_INPUT` - the `irq_num` exceeds The maximum supported number of external interrupts.
- `TFM_HAL_ERROR_GENERIC` - failed to disable the interrupt.
- `TFM_HAL_SUCCESS` - the interrupt is successfully disabled.

tfm_hal_irq_clear_pending()

Prototype

```
enum tfm_hal_status_t tfm_hal_irq_clear_pending(uint32_t irq_num)
```

Description

This API clears an active and pending interrupt.

Parameter

- `irq_num` - the interrupt to be disabled with a number

Return Values

- `TFM_HAL_ERROR_INVALID_INPUT` - the `irq_num` exceeds The maximum supported number of external interrupts.
- `TFM_HAL_ERROR_GENERIC` - failed to clear the pending interrupt.
- `TFM_HAL_SUCCESS` - the pending interrupt has been cleared.

Initializations

Prototype

```
enum tfm_hal_status_t {source_symbol}_init(void *p_pt,
                                           const struct irq_load_info_t *p_ildi)
```

The `{source_symbol}` is:

- `irq_{source}`, if the `source` attribute of the IRQ in Partition manifest is a number
- Lowercase of `source` attribute, if `source` is a symbolic name

Description

Each interrupt has an initialization function individually. The *SPM* calls the functions while loading the Partitions.

The following initializations are required for each interrupt:

- Setting the priority. The value must between 0 to 0x80 exclusively.
- Ensuring that the interrupt targets the Secure State.
- Saving the input parameters for future use.

Platforms can have extra initializations as well.

Parameter

- `p_pt` - pointer to Partition runtime struct of the owner Partition
- `p_ildi` - pointer to `irq_load_info_t` struct of the interrupt

Note

Please refer to the :doc: *IRQ intergration guide<tfm_secure_irq_integration_guide>* for more information.

API Definition for Secure Partitions

The Secure Partition (SP) *HAL* mainly focuses on two parts:

- Peripheral devices. The peripherals accessed by the *TF-M* default Secure Partitions.
- Secure Partition abstraction support. The Secure Partition data that must be provided by the platform.

The Secure Partition abstraction support will be introduced in the peripheral API definition.

ITS and PS flash API

There are two kinds of flash:

- Internal flash. Accessed by the PSA Internal Trusted Storage (ITS) service.
- External flash. Accessed by the PSA Protected Storage (PS) service.

The ITS HAL for the internal flash device is defined in the `tfm_hal_its.h` header and the PS HAL in the `tfm_hal_ps.h` header.

Macros

Internal Trusted Storage

TFM_HAL_ITS_FLASH_DRIVER

Defines the identifier of the CMSIS Flash ARM_DRIVER_FLASH object to use for ITS. It must have been allocated by the platform and will be declared extern in the HAL header.

TFM_HAL_ITS_PROGRAM_UNIT

Defines the size of the ITS flash device's physical program unit (the smallest unit of data that can be individually programmed to flash). It must be equal to TFM_HAL_ITS_FLASH_DRIVER.GetInfo()->program_unit, but made available at compile time so that filesystem structures can be statically sized.

TFM_HAL_ITS_FLASH_AREA_ADDR

Defines the base address of the dedicated flash area for ITS.

TFM_HAL_ITS_FLASH_AREA_SIZE

Defines the size of the dedicated flash area for ITS in bytes.

TFM_HAL_ITS_SECTORS_PER_BLOCK

Defines the number of contiguous physical flash erase sectors that form a logical filesystem erase block.

Protected Storage

TFM_HAL_PS_FLASH_DRIVER

Defines the identifier of the CMSIS Flash ARM_DRIVER_FLASH object to use for PS. It must have been allocated by the platform and will be declared extern in the HAL header.

TFM_HAL_PS_PROGRAM_UNIT

Defines the size of the PS flash device's physical program unit (the smallest unit of data that can be individually programmed to flash). It must be equal to TFM_HAL_PS_FLASH_DRIVER.GetInfo()->program_unit, but made available at compile time so that filesystem structures can be statically sized.

TFM_HAL_PS_FLASH_AREA_ADDR

Defines the base address of the dedicated flash area for PS.

TFM_HAL_PS_FLASH_AREA_SIZE

Defines the size of the dedicated flash area for PS in bytes.

TFM_HAL_PS_SECTORS_PER_BLOCK

Defines the number of contiguous physical flash erase sectors that form a logical filesystem erase block.

Optional definitions

The `TFM_HAL_ITS_FLASH_AREA_ADDR`, `TFM_HAL_ITS_FLASH_AREA_SIZE` and `TFM_HAL_ITS_SECTORS_PER_BLOCK` definitions are optional. If not defined, the platform must implement `tfm_hal_its_fs_info()` instead.

Equivalently, `tfm_hal_its_ps_info()` must be implemented by the platform if `TFM_HAL_ITS_FLASH_AREA_ADDR`, `TFM_HAL_ITS_FLASH_AREA_SIZE` or `TFM_HAL_ITS_SECTORS_PER_BLOCK` is not defined.

Objects

ARM_DRIVER_FLASH

The ITS and PS HAL headers each expose a CMSIS Flash Driver instance.

```
extern ARM_DRIVER_FLASH TFM_HAL_ITS_FLASH_DRIVER

extern ARM_DRIVER_FLASH TFM_HAL_PS_FLASH_DRIVER
```

The CMSIS Flash Driver provides the flash primitives ReadData(), ProgramData() and EraseSector() as well as GetInfo() to access flash device properties such as the sector size.

Types

tfm_hal_its_fs_info_t

Struct containing information required from the platform at runtime to configure the ITS filesystem.

```
struct tfm_hal_its_fs_info_t {
    uint32_t flash_area_addr;
    size_t flash_area_size;
    uint8_t sectors_per_block;
};
```

Each attribute is described below:

- `flash_area_addr` - Location of the block of flash to use for ITS
- `flash_area_size` - Number of bytes of flash to use for ITS
- `sectors_per_block` - Number of erase sectors per logical FS block

tfm_hal_ps_fs_info_t

Struct containing information required from the platform at runtime to configure the PS filesystem.

```
struct tfm_hal_ps_fs_info_t {
    uint32_t flash_area_addr;
    size_t flash_area_size;
    uint8_t sectors_per_block;
};
```

Each attribute is described below:

- `flash_area_addr` - Location of the block of flash to use for PS
- `flash_area_size` - Number of bytes of flash to use for PS
- `sectors_per_block` - Number of erase sectors per logical FS block

Functions

tfm_hal_its_fs_info()

Prototype

```
enum tfm_hal_status_t tfm_hal_its_fs_info(struct tfm_hal_its_fs_info_t *fs_info);
```

Description

Retrieves the filesystem configuration parameters for ITS.

Parameter

- `fs_info` - Filesystem config information

Return values

- `TFM_HAL_SUCCESS` - The operation completed successfully
- `TFM_HAL_ERROR_INVALID_INPUT` - Invalid parameter

Note

This function should ensure that the values returned do not result in a security compromise. The block of flash supplied must meet the security requirements of Internal Trusted Storage.

tfm_hal_ps_fs_info()

Prototype

```
enum tfm_hal_status_t tfm_hal_ps_fs_info(struct tfm_hal_ps_fs_info_t *fs_info);
```

Description

Retrieves the filesystem configuration parameters for PS.

Parameter

- `fs_info` - Filesystem config information

Return values

- `TFM_HAL_SUCCESS` - The operation completed successfully
- `TFM_HAL_ERROR_INVALID_INPUT` - Invalid parameter

Note

This function should ensure that the values returned do not result in a security compromise.

11.4.3 Cooperative Scheduling Rules

Author
Ashutosh Singh

Organization
Arm Limited

Contact
Ashutosh Singh <ashutosh.singh@arm.com>

TF-M Scheduler - Rules

On ArmV8-M CPUs, NSPE and SPE share the same physical processing element(PE). A TF-M enabled systems need to be able to handle asynchronous events (interrupts) regardless of current security state of the PE, and that may lead to scheduling decisions. This introduces significant complexity into TF-M. To keep the integrity of (NSPE and SPE) schedulers and call paths between NSPE and SPE, following set of rules are imposed on the TF-M scheduler design.

Objectives and Requirements

1. Decoupling of scheduling decisions between NSPE and SPE
2. Efficient interrupt handling by SPE as well as NSPE
3. Reduce critical sections on the secure side to not block interrupts unnecessarily
4. Scalability to allow simplification/reduction of overheads to scale down the design for constrained devices
5. Reduce impact on NSPE software design

a. NSPE interrupt handling implementation should be independent

b. Impact on the NSPE scheduler should be minimal

c. No assumptions should be made about NSPE's scheduling capabilities

Scheduler Rules for context switching between SPE and NSPE

To allow coherent cooperative scheduling, following set of rules are imposed on security state changes. The switching between SPE and NSPE can be triggered in multiple ways.

Involuntary security state switch; when the software has no control over the switch:

- A NSPE interrupt takes control into NSPE from SPE
- A SPE interrupt takes control into SPE from NSPE

Voluntary security state switch; when software programmatically makes the switch:

- A NSPE exception handler returns from NSPE to pre-empted SPE context
- A SPE exception handler returns from SPE to pre-empted NSPE context
- NSPE makes a function call into SPE
- SPE returns a call from NSPE
- SPE makes a function call into NSPE (not covered in current design)
- NSPE returns a call from SPE (not covered in current design)

In order to maintain the call stack integrity across NSPE and SPE, following rules are imposed on all security state switches.

Rules for NSPE Exception handling

1. **The NSPE exception handler is allowed to trigger a NSPE context switch (regardless of security state of the preempted context.**

This is expected behaviour of any (RT)OS.

2. **The NSPE scheduler must eventually 'restore' the preempted (by exception) context.**

This is expected behaviour of any (RT)OS.

3. **If NSPE exception results in a NSPE context switch, SPM must be informed of the scheduling decision; this must be done BEFORE the execution of newly scheduled-in context.**

This is to ensures integrity of the call stack when SPE is ready to return a previous function call from NSPE.

Rules for SPE Exception handling

1. **All of the SPE interrupts must have higher priority than NSPE interrupts**

This is rule is primarily for simplifying the SPM design.

2. **The SPE interrupt handler is allowed to trigger a SPE context switch (regardless of security state of the pre-empted context)**

If the SPE context targeted by the interrupt is not same as current SPE context, the SPM may choose to switch the current running SPE context based on priority.

3. **SPE scheduler must treat pre-empted context as one of the SPE contexts**

   a. If the pre-empted SPE context is SP1, the TCB for SP1 should be used for saving the context. i.e. the context of SP1 should be saved before scheduling anything other secure partition.

   b. If SP1 was pre-empted by a NSPE interrupt, and subsequent NSPE execution is pre-empted by SPE exception (before NSPE scheduling decision is communicated back to SPM) – SP1 TCB must be used for saving the context In this case SPM is not yet aware of the NSPE context switch, from SPM's standpoint SP1 is still executing, so SPM assumes that the preempted context is SP1.

   c. If SP1 was pre-empted by a NSPE interrupt, and subsequent NSPE execution is pre-empted by SPE exception *after* NSPE scheduling decision is communicated back to SPM) - a TCB dedicated to NSPE should be used for saving the context.

   When NSPE scheduler communicates the scheduling decision to SPM, SPM must save the SP1 context, if a SPE interrupt preempts the currently running NSPE context, SPM should save the context to a dedicated NSPE TCB.

   d. The SPE scheduler must eventually 'restore' the pre-empted context. This is an expected behaviour of any scheduler.

4. **All of the interrupts belonging to a partition must have same priority.**

This serializes ISR execution targeted for same partition.

5. **In case of nested interrupts, all of the ISRs must run to finish before any service code is allowed to run**

This is an expected behaviour of any scheduler.

6. **If the previously preempted context was a NSPE ISR context, SPE ISR handler must return to preempted NSPE context.**

This is an expected behaviour of any scheduler to return to preempted context.

Rules for NSPE to SPM function call (and NSPE scheduler)

1. Current NSPE context must have been communicated to SPM, otherwise SPM cannot guarantee NSPE function calling stack integrity.

Rules for Function Return from SPE to NSPE with result

1. **The result available on SPE side are for currently active NSPE context.**

To maintain call stack integrity, if SPE is ready to return to NSPE, it can do function return only if the SPE return path corresponds to currently active NSPE context.

2. **Last entry into secure world happened programmatically (Voluntary security state switch into SPE)**

i.e. control is voluntarily given back by NSPE, either through a function call, or a context restore via 'return to SPE from NSPE'. As opposed to a SPE interrupt bringing back the execution into SPE.

3. **The current NSPE call stack has not already been returned with SPM_IDLE.**

This rule applies if following optional feature is enabled.

Rules for Return from SPE to NSPE with SPM_IDLE

This is optional part of the design as it introduces significant complexity on both sides of the security boundary. It allows yielding of the CPU to NSPE when SPE has not CPU execution to do but it has not yet finished the previous request(s) from NSPE; i.e. SPE is waiting on arrival of a SPE interrupt.

1. **Last entry into secure world happens programmatically (Voluntary security context switch into SPE)**

i.e. control is voluntarily given back by NSPE, either through a function call, or a context restore via 'return to SPE from NSPE'. As opposed to a SPE interrupt bringing back the execution into SPE.

2. **The result for the currently active NSPE entity is not yet available, the called service is waiting (on interrupt/event).**

SPE request corresponding to currently active NSPE caller is not yet completed and is waiting on an ISR.

3. **The current NSPE call stack has not already been returned with SPM_IDLE.**

Rules for NSPE pend irq based return from SPE to NSPE

This is optional part of the design as it introduces significant complexity on both sides. This works in conjunction with [Rules for Return from SPE to NSPE with SPM_IDLE](#rules-for-return-from-spe-to-nspe-with-spm_idle). In this scenario, when SPE is ready with result for a previous call from NSPE, it raises a pended IRQ to NSPE instead of returning the function call path.

1. **The SPE has finished a NSPE request.**
2. **The corresponding NSPE context has already been returned with SPM_IDLE.**

Rules for ISR pre-emption

1. **A higher priority NSPE interrupt is allowed to preempt a lower priority NSPE ISR**
2. **A higher priority SPE interrupt is allowed to preempt a lower priority SPE ISR**
3. **A SPE interrupt is allowed to preempt NSPE ISR**
4. **A NSPE interrupt is not allowed to preempt SPE ISR**
5. **All interrupts belonging to a service must have same priority**

11.4.4 Code Generation With Jinja2

Author
Mate Toth-Pal

Organization
Arm Limited

Contact
Mate Toth-Pal <mate.toth-pal@arm.com>

Generating files from templates in TF-M

Some of the files in TF-M are generated from template files. The files to be generated are listed in `tools/tfm_generated_file_list.yaml`. For each generated file `<path_to_file>/<filename>` the template file is `<path_to_file>/<filename>.template`. The templates are populated with partition information from the partition manifests. The manifests to be used for the generation are listed in `tools/tfm_manifest_list.yaml`.

Custom generator script - Current method

`tools/tfm_parse_manifest_list.py` Python script is used to generate files from the templates. This script calls the `tools/generate_from_template.py` to parse the template files, and uses `tools/keyword_substitution.py` to substitute the keychains with actual values from the manifest files.

Use Jinja2 template engine - proposal

The proposal is to eliminate the template parser and substituter scripts, and call the Jinja2 template engine library from `tools/tfm_parse_manifest_list.py` to do the substitution.

More information on jinja2: http://jinja.pocoo.org/

Changes needed:

- `tools/tfm_parse_manifest_list.py` have to be modified to call the Jinja2 library instead of the custom scripts. The data structure required by the library is very similar to the one required by the current scripts.
- template files needs to be rewritten to the Jinja syntax: The control characters to be replaced (like `@!` -> `{%`) and `for` statements to be added to iterate over the substitutions.

Improvements over the current solution

- Compatible with Python 2.7 and Python 3, while current solution only supports Python 2.7
- More advanced functionality: direct control over the list of items for a keychain, meta information on that list (like length)
- Well documented (see website)
- Jinja2 is free and open-source software, BSD licensed, just like TF-M. It is a well established software in contrast with the current proprietary solution.

Example

Below code snippet enumerates services in Secure Partitions:

```
{% for partition in partitions %}
    {% if partition.manifest.services %}
        {% for service in partition.manifest.services %}
            {do something for the service}
        {% endfor %}
    {% endif %}
{% endfor %}
```

11.4.5 Fixing implicit casting for C enumeration values

Author
Hugues de Valon

Organization
Arm Limited

Contact
hugues.devalon@arm.com

Abstract

C enumerations provide a nice way to increase readability by creating new enumerated types but the developer should take extra care when mixing enumeration and integer values. This document investigates C enumerations safety and proposes strategies on how to fix the implicit casting of the enumeration values of TF-M with other types.

C99 Standard point of view

In TF-M many implicit casts are done between integer types (`uint32_t`, `int32_t`, `size_t`, etc), enumerated types (`enum foobar`) and enumeration constants (`FOOBAR_ENUM_1`).

According to the C99 standard[1]:

> **§6.2.5, 16**: An enumeration comprises a set of named integer constant values. Each distinct enumeration constitutes a different numerated type.
>
> **§6.7.2.2, 2**: The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an int.
>
> **§6.7.2.2, 3**: The identifiers in an enumerator list are declared as constants that have type int and may appear wherever such are permitted.
>
> **§6.7.2.2, 4**: Each enumerated type shall be compatible with char, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined, but shall be capable of representing the values of all the members of the enumeration.

From these four quotes from the C99 standard[Page 435, 1], the following conclusions can be made:

- an enumeration defines a new type and should be treated as such
- the enumeration constants must only contains value representable as an `int`
- the enumeration constants have type `int`
- the actual type of the enumeration can be between `char`, signed and unsigned `int`. The compiler chooses the type it wants among those that can represent all declared constants of the enumeration.

Example:

```
enum french_cities {
    MARSEILLE,
    PARIS,
    LILLE,
    LYON
};
```

In that example, `MARSEILLE`, `PARIS`, `LILLE` and `LYON` are enumeration constants of type `int` and `enum french_cities` is a enumerated type which can be of actual type `char`, `unsigned int` or `int` (the compiler chooses!).

For these reasons, doing an implicit cast between an enumeration and another type is the same as doing an implicit cast between two different types. From a defensive programming point of view, it should be checked that the destination type can represent the values from the origin type. In this specific case it means four things for enumerations:

- it is always safe to assign an enumeration constant to an int, but might be better to cast to show intent.
- when casting an enumeration constant to another type, it should be checked that the constant can fit into the destination type.

[1] C99 standard: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

- when casting from an integer type (`uint32_t`, `int32_t`, etc) to an enumeration type, it should be checked that the integer's value is one of the enumeration constants. The comparison needs to be done on the biggest type of the two so that no information is lost. C integer promotion should automatically do that for the programmer (check §6.3.1.8, 1 for the rules).
- when casting from an enumeration type to an integer type, it should be checked that the enumeration type value fits into the integer type. The value of a variable which has the type of an enumeration type is not limited to the enumeration constants of the type. An enumeration constant will always fit into an `int`.

Strategies to fix

0. Replace the enumerated type with an integer type and replace the enumeration constant with preprocessor constants.
1. Whenever possible, try to use matching types to avoid implicit casting. It happens, for example, for arithmetic operations, function calls and function returns. This strategy always have the lowest performance impact.
2. When using an enumeration constant in an arithmetic operation with another type, verify that the constant can fit in the other type and cast it.
3. When converting an integer to an enumeration type, use a conversion function to check if the integer matches an enumeration constant. To not impact performance too much, this function should be an inline function. If it does not match, use (or add) the error constant or return an error value.
4. When converting an enumeration type to an integer, use a conversion function to check that the integer type can contain the enumeration value.

Design proposal for TF-M

In TF-M, an action will be taken for all enumerated types and enumeration constants that are used for implicit casting. The goal of this proposal is to remove all implicit casting of enumeration values in TF-M.

The following enumerated types will be removed and replaced with preprocessor constants (strategy 0). These enumerated types are not used in TF-M but only the constants they declare.

- `enum spm_part_state_t`
- `enum spm_part_flag_mask_t`
- `enum tfm_partition_priority`

The following enumerated types will be kept because they are used in the prototypes of functions and are useful for debugging. Whenever possible, strategy 1 will be applied to remove implicit casting related with those enumerations but dynamic conversions will be used if the first option would create too much change in the code base.

- `enum tfm_status_e`: the return type of the following functions will be changed to return the `enum tfm_status_e` type. These functions are already returning the enumeration constants, but implicitly casted to an integer type like `int32_t`.
  - `int32_t check_address_range`
  - `int32_t has_access_to_region`
  - `int32_t tfm_core_check_sfn_parameters`
  - `int32_t tfm_start_partition`
  - `int32_t tfm_return_from_partition`
  - `int32_t tfm_check_sfn_req_integrity`
  - `int32_t tfm_core_check_sfn_req_rules`

  - int32_t tfm_spm_sfn_request_handler
  - int32_t tfm_spm_sfn_request_thread_mode

- enum tfm_buffer_share_region_e: the following function prototypes will be changed:
  - tfm_spm_partition_set_share(uint32_t partition_idx, uint32_t share) -> tfm_spm_partition_set_share(uint32_t partition_idx, enum tfm_buffer_share_region_e share)
- enum tfm_memory_access_e
- enum attest_memory_access_t
- enum engine_cipher_mode_t
- mbedtls_cipher_type_t

The following enumerated types are used for error code values of Secure service calls. They should be kept as they are part of the interface and might be used by external parties in Non-Secure code. For the Initial Attestation service, the enumeration is defined in the PSA Attestation API specifications.

- enum psa_attest_err_t
- enum psa_audit_err
- enum tfm_platform_err_t

Implicit casting related with these enumerations is happening in two locations of TF-M and need conversion functions in those locations, because the types can not be changed:

- In the Non-Secure Client library, all of the Secure Service functions implicitly cast the uint32_t returned by tfm_ns_lock_dispatch to these enumerated types. Strategy 3 is needed here.
- In all of the veneer functions, there is an implicit cast from the int32_t value returned by the SVC request function (tfm_core_*_request) to these enumerated types. Strategy 3 is needed here as well. The implicit cast will eventually be removed if all of the services are using the Uniform Signatures Prototypes so that the veneer functions all return psa_status_t which is an int32_t.

If the interface of those services can be changed, these enumerations could be removed and replaced with the psa_status_t type to remove the implicit casting.

11.5 FF-M Isolation

Organization
Arm Limited

Contact
tf-m@lists.trustedfirmware.org

This document analyzes the isolation rules of implementing FF-M 1.0 isolation and introduces the reference implementation in TF-M, which complies the rules by operating the hardware and software resources.

Note: Reference the document *Glossary* for terms and abbreviations.

11.5.1 Introduction

This chapter describes the definitions from `FF-M` and analyzes the possible implementation keypoints.

Isolation Levels

There are 3 isolation levels (1-3) defined in `FF-M`, the greater level number has more isolation boundaries.

The definition for Isolation Level 1:

- L1.1 NPSE needs protection from nobody.
- L1.2 SPE needs protection from NSPE.

The definition for Isolation Level 2:

- L2.1 NPSE needs protection from nobody.
- L2.2 Application Root of Trust (ARoT) needs protection from NSPE.
- L2.3 PSA Root of Trust (PRoT) needs protection from NSPE and ARoT.

The definition for Isolation Level 3:

- L3.1 NPSE needs protection from nobody.
- L3.2 Secure Partition needs protection from NSPE and other Secure Partitions.
- L3.3 PSA Root of Trust (RoT) domain needs protection from NSPE and all Secure Partitions.

Important: A Secure Partition RoT Service is a Root of Trust Service implemented within a Secure Partition. An Application RoT Service must be implemented as a Secure Partition RoT Service. But it is implementation-defined whether a PSA RoT Service is a Secure Partition RoT Service.

Here listed several possible PSA RoT Service implementation mechanisms:

1. Implement PSA RoT Services in Secure Partitions with respective boundaries.
2. Implement PSA RoT Services in Secure Partitions, but no boundaries between these Secure Partitions as they are in the PSA RoT Domain.
3. Implement PSA RoT Services in a customized way instead of Secure Partitions, an internal library of PSA RoT domain e.g.

TF-M chooses the 2nd option to balance performance and complexity.

Isolation Rules

The essence of isolation is to protect the assets of one protection domain from being accessed from other domains. The isolation levels define where the isolation boundaries should be placed, the isolation rules define the strength of the isolation the boundaries should offer.

Note: Refer to chapter *Memory Assets* in Firmware Framework for M (FF-M) to know asset class items. Assets are represented by memory addresses in the system memory map, which makes assets named *Memory Assets*. The often-seen asset items are ROM, RAM, and memory-mapped peripherals.

Memory Asset Class

There are 3 memory asset classes defined in Firmware Framework for M (FF-M):

- Code
- Constant data
- Private data

There are 6 isolation rules for protecting assets. Here lists the simplified description for the rules, check chapter `3.1.2` of `FF-M 1.0` for the original description:

- I1. Only Code is executable.
- I2. Only private data is writable.
- I3. If domain A needs protection from domain B, then Private data in domain A cannot be accessed by domain B.
- I4. (Optional) If domain A needs protection from domain B, then Code and Constant data in domain A is not readable or executable by domain B.
- I5. (Optional) Code in a domain is not executable by any other domain.
- I6. (Optional) All assets in a domain are private to that domain and cannot be accessed by any other domain, with the following exception: The domain containing the SPM can only access Private data and Constant data assets of other domains when required to implement the PSA Firmware Framework API.
- I7. (Optional, added in FF-M 1.1) Constant data is not executable.

  The first 3 rules from `I1` to `I3` defines the mandatory rules to comply the isolation, while `I4` to `I6` are optional rules to enhance the isolation boundaries.

  Important: There is a table in the chapter `3.1.2` of `FF-M 1.0` under `I1` lists the asset types and allowed access method. Preventing executable access on constant data costs more hardware resources, so there is an optional rule I7 created in FF-M Extensions (FF-M 1.1) to comfort implementations with constrained hardware resources.

Hardware Infrastructure

To implement a secure system, the hardware security framework (TrustZone or multiple-core e.g.) and their auxiliary components (SAU e.g.) are required to ensure the isolation between SPE and NSPE. The requirements:

Important: The interface between secure and non-secure states needs to be fully enumerated and audited to prove the integrity of the secure state isolation.

Besides this SPE and NSPE isolation mechanism, the following analyzes the implementation rules to find out the hardware requirements for isolation inside SPE domains:

- I1 and I2: The assets can be categorized into 3 *Memory Asset Class*, each type has the specific access rules.
- I3: The private data access from the prevented domain needs to be blocked.
- I4: All the assets access from the prevented domain needs to be blocked.
- I5: Code execution from all other domains (even the domain not prevented from) needs to be blocked.

- I6: All the assets access from all other domains (includes non-prevented domain) needs to be blocked, but, SPM is an exception, which can access the private data and constant data of the current domain.

The above items list the requirements for memory access, here are two more points:

- If the memory device or the peripheral are shared between multiple hosts (Such as multiple CPU or DMA, etc), specific hardware protection units need to be available for validating accesses to that device or peripheral.
- The MMIO range for Secure Partitions is not allowed to be overlapped, which means each partition should have exclusive memory-mapped region if they require a peripheral device. The memory-mapped region is regarded as the private data so access to this area needs to be validated.

11.5.2 Reference Implementation

This chapter describes the isolation implementation inside SPE by using the Armv8m architecture component - Memory Protection Unit (MPU). The MPU can isolate CPU execution and data access.

Note: Previous version M-profile architecture MPU setting is similar in concept but the difference in practical register formats, which is not described in this document.

The MPU protects memory assets by regions. Each region represents a memory range with specific access attributes.

Note: The maximum numbers of MPU regions are platform-specific.

The SPM is running under the privileged mode for handling access from services. The MPU region for SPM needs to be available all the time since SPM controls the MPU setting while scheduling.

Since partitions are scheduled by SPM, the MPU regions corresponding to the partitions can be configured dynamically while scheduling. Since there is only one running at a time and all others are deactivated, the SPM needs to set up necessary regions for each asset type in one partition only.

There is re-usable code like the C-Runtime and RoT Service API which are same across different partitions. TF-M creates a Secure Partition Runtime Library (SPRTL) as a specific library shared by the Secure Partition. Please refer to *Secure Partition Runtime Library* for more detail.

Note: Enable SPRTL makes it hard to comply with the rules I4, I5 and I6, duplicating the library code can be one solution but it is not “shared” library anymore.

As mentioned in the last chapter, MMIO needs extra MPU regions as private data.

MPU Region Access Permission

The following content would mention the memory access permission to represent the corresponded asset classes.

These access permissions are available on Armv8m MPU:

- Privileged Read-Only (RO)
- All RO
- Privileged Read-Write (RW)
- All RW
- Execution Never (XN)

And one more Armv8.1M access permssion:

- Privileged Execution Never (PXN)

The available regions type list:

Type	Attributes	Privilege Level	Asset
P_RO	RO	Privileged	PRoT Code
P_ROXN	RO + XN	Privileged	PRoT Constant Data
P_RWXN	RW + XN	Privileged	PRoT Private Data/Peripheral
A_RO	RO	Any privilege	Partition/SPRTL Code
A_ROXN	RO + XN	Any privilege	Partition/SPRTL Constant Data
A_RWXN	RW + XN	Any privilege	Partition/SPRTL Private Data/Peripheral
A_ROPXN	RO + PXN	Any privilege	Armv8.1M Partition/SPRTL Code

Example Image Layout

The secure firmware image contains components such as partitions, SPM and the shared code and data. Each component may have different class assets. There would be advantages if placing the assets from all components with the same access attributes into one same region:

- The data relocating or clearing when booting can be done in one step instead of breaking into fragments.
- Assets with statically assigned access attribute can share the same MPU region which saves regions.

Take the TF-M existing implementation image layout as an example:

```
 Level 1      Level 2             Level 3
 Boundaries   Boundaries          Boundaries
+------------+----------+------------------------------------+
|            |          |    PRoT SPM                Code    |
|            | PRoT     +------------------------------------+
|            | Code     |    PRoT Service            Code    |
|  Code      +----------+------------------------------------+
|  (ROM)     |          |    Partition 1             Code    |
|            |          +------------------------------------+
|            | ARoT     |    Partition N             Code    |
|            | Code     +------------------------------------+
|            |          |    SPRTL                   Code    |
+------------+----------+------------------------------------+
Check [4] for more details between Code and Constant Data.
+------------+----------+------------------------------------+
|            | PRoT     |    PRoT SPM      Constant Data    |
|            | Constant +------------------------------------+
|            | Data     |    PRoT Service  Constant Data    |
|  Constant  +----------+------------------------------------+
|   Data     | ARoT     |    Partition 1   Constant Data    |
|   (ROM)    | Constant +------------------------------------+
|            | Data     |    Partition N   Constant Data    |
|            |          +------------------------------------+
|            |          |    SPRTL         Constant Data    |
+------------+----------+------------------------------------+


+------------+----------+------------------------------------+
```

(continues on next page)

(continued from previous page)

```
|            | PRoT     |    PRoT SPM        Private Data    |
|            | Private  +------------------------------------+
|            | Data     |    PRoT Service    Private Data    |
|  Private   +----------+------------------------------------+
|   Data     |          |    Partition 1     Private Data    |
|   (RAM)    | ARoT     +------------------------------------+
|            | Private  |    Partition N     Private Data    |
|            | Data     +------------------------------------+
|            |          |    SPRTL           Private Data    |
+------------+----------+------------------------------------+
```

Note:

1. Multiple binaries image implementation could also reference this layout if its hardware protection unit can cover the exact boundaries mentioned above.
2. Private data includes both initialized and non-initialized (ZI) sections. Check chapter `3.1.1` of `FF-M` for the details.
3. This diagram shows the boundaries but not orders. The order of regions inside one upper region can be adjusted freely.
4. As described in the `important` of *Memory Asset Class*, the setting between Code and Constant Data can be skipped if the executable access method is not applied to constant data. In this case, the groups of Code and Constant Data can be combined or even mixed – but the boundary between PRoT and ARoT are still required under level higher than 1.

Example Region Numbers under Isolation Level 3

The following table lists the required regions while complying the rules for implementing isolation level 3. The level 1 and level 2 can be exported by simplifying the items in level 3 table.

Important: The table described below is trying to be shared between all supported platforms in Trusted Firmware - M. It is obvious that some platforms have special characteristics. In that case, the specific layout table for a particular platform can be totally redesigned but need to fulfil the isolation level requirements.

- Care only the running partitions assets since deactivated partition does not need regions.
- *X* indicates the existence of this region can't comply with the rule.
- An $ATTR + n$ represent extra `n` regions are necessary.
- Here assumes the rules with a greater number covers the requirements in the rules with less number.

Here lists the required regions while complying with the rules:

Region Purpose	I1 I2 I3	I4	I5	I6
PRoT SPM Code	A_RO	P_RO	P_RO	P_RO
PRoT Service Code				A_ROPXN
Active Partition Code		A_RO	A_ROPXN	
SPRTL Code		X	X	X
PRoT SPM RO	A_ROXN	P_ROXN	P_ROXN	P_ROXN
PRoT Service RO				A_ROXN
Active Partition RO		A_ROXN	A_ROXN	
SPRTL RO		X	X	X
PRoT SPM RW	P_RWXN	P_RWXN	P_RWXN	P_RWXN
PRoT Service RW				A_RWXN
Active Partition RW	A_RWXN	A_RWXN	A_RWXN	
SPRTL RW [5]	A_RWXN + 1	X	X	X
Partition Peri	A_RWXN + n	A_RWXN + n	A_RWXN + n	A_RWXN + n
Total Numbers	[1]	[2]	[3]	[4]

Note:

1. Total number = A_RO + A_ROXN + P_RWXN + A_RWXN + (1 + n)A_RWXN, while n equals the maximum peripheral numbers needed by one partition. This is the configuration chosen by the reference implementation.
2. Total number = P_RO + P_ROXN + P_RWXN + A_RO + A_ROXN + (1 + n)A_RWXN, the minimal result is *6*, and SPRTL can not be applied.
3. Total number = P_RO + P_ROXN + P_RWXN + A_ROXN + (1 + n)A_RWXN + A_ROPXN, the minimal result is *6*, SPRTL can not be applied, and PXN is required.
4. Total number = P_RO + P_ROXN + P_RWXN + A_ROXN + (1 + n)A_RWXN + A_ROPXN, the minimal result is *6*, SPRTL can not be applied, and PXN is required. To comply with this rule, the PSA RoT Service needs to be implemented as Secure Partitions.
5. This data belongs to SPRTL RW but it is set as Read-Only and only SPM can update this region with the activate partition's metadata for implementing functions with owner SP's context, such as heap functions. This region can be skipped if there is no metadata required (such as no heap functionalities required).

The memory-mapped regions for peripherals have different memory access attributes in general, they are standalone regions in MPU even their attributes covers 'A_RWXN'.

Default access rules

Hardware protection components MAY have the capability to collect regions not explicitly configured in static or runtime settings, and then apply default access rules to the collected. Furthermore, one default rule can be applied to multiple non-contiguous regions which makes them share a common boundary. This operation sets up a standalone 'region' as same as other explicitly configured regions. And it doesn't affect the analysis summary above - just be aware that some regions listed in the table MAY not be explicitly configured.

Take the MPU as an example, MPU can assign a default privileged access attribute to the regions (SPM and PRoT regions e.g.) not explicitly configured. This feature can reduce required MPU regions and ease the programming because regions can be put non-address-contiguous and skip the explicit configuration.

Important: When this default access rules mechanism is applied, the non-explicitly-expressed regions must be reviewed to ensure the isolation boundaries are set properly.

Interfaces

The isolation implementation is based on the HAL framework. The SPM relies on the HAL API to perform the necessary isolation related operations.

The requirement the software need to do are these:

- Create enough isolation protection at the early stage of system booting, just need to focus on the SPM domain.
- Create an isolation domain between secure and non-secure before the jump to the non-secure world.
- Create an isolation domain for each Secure Partition after the Secure Partition is loaded and before jumping to its entry point. The isolation domain should cover all the assets of the Secure Partition, include all its memory, interrupts, and peripherals.
- Switch isolation domains when scheduling different Secure Partitions.
- It is also a requirement that the platform needs to help to check if the caller of the PSA APIs is permitted to access some memory ranges.

The design document *TF-M Hardware Abstraction Layer* gives a detail design, include the platform initialization, isolation interfaces. Please refer to it for more detail.

Appendix

Firmware Framework for M (FF-M)

FF-M Extensions (FF-M 1.1)

Trusted Base System Architecture for M (TBSA-M)

11.6 TF-M builtin keys

Author
Raef Coles

Organization
Arm Limited

Contact
raef.coles@arm.com

11.6.1 Introduction

When TF-M is integrated on a platform, the platform itself can provide several keys for usage which are bound to the device instead of being owned by a specific secure partition. When those keys are readable by the secure processing environment, the platform must provide a function to load these keys in the HAL layer, either loading them from OTP or another platform specific location or subsystem. These keys are henceforth referred to as “builtin keys”, and might include (but are not limited to) the following:

1. The Hardware Unique Key (HUK)
2. The Initial Attestation Key (IAK)

The `tfm_builtin_key_loader` component implements a mechanism to discover those keys and make them available to the PSA Crypto APIs for usage. Note that if a key is stored in a subsystem which can’t be read by the secure processing environment, a full opaque driver[1] must be used to be able to use those keys through the PSA Crypto APIs. This document focuses only on the case where the keys can be read by the SPE (i.e. running TF-M), so the driver applies only on those cases, which can be considered as “_transparent_ builin keys”.

In TF-M’s legacy solution, the IAK is loaded by the attestation service as a volatile key, which requires some key-loading logic to be implemented by that partition. The HUK is not loaded in the crypto service, and is instead used by an implementation of a TF-M specific KDF algorithm which then loads the key and invokes Mbed TLS directly. Both solutions are far from ideal as they require an effort to load (and duplicate) the keys per each user, or require a dedicated code in TF-M that directly interacts with the HAL layer and invokes functions from the crypto library bypassing the PSA Crypto interface. The aim of the `tfm_builtin_key_loader` driver is to provide a uniform interface to use the builtin keys available in a platform.

Implementing a driver to deal with builtin keys allows to expand the legacy solution for dealing with this type of keys in several ways. For example, it avoids the need for partitions to implement dedicated mechanisms for probing the HAL layer for builtin keys and load them as volatile. It removes the need to have implementation specific call flows for deriving keys from the HUK (as that requirement now is fulfilled by the driver itself transparently). It allows uniform access to the keys also from the NS world (in any case, subject to the policy dictated by the platform). Correctly abstracts away details of the key handling mechanism from TF-M partitions into the PSA Crypto core key management subsystem.

11.6.2 PSA builtin keys

The PSA Cryptographic API provides a mechanism for accessing keys that are stored in platform-specific locations (often hardware accelerators or OTP). A builtin key is assigned a specific key_id, i.e. a handle, which is hardcoded at build time and must be selected in the range [MBEDTLS_PSA_KEY_ID_BUILTIN_MIN, MBEDLTS_PSA_KEY_ID_BUILTIN_MAX]. A user of the PSA Crypto API can reference those keys directly by using these handles. It is up to the platform to specify policy restrictions for specific users of the keys, e.g. an NS entity, or a secure partition. The PSA Crypto core will then check those policies to grant or deny access to the builtin key for that user.

[1] PSA cryptoprocessor driver interface: https://github.com/Mbed-TLS/mbedtls/blob/development/docs/proposed/psa-driver-interface.md

11.6.3 PSA cryptoprocessor driver API

The PSA specification allows the PSA Crypto APIs to defer their operation to an accelerator driver, through a mechanism described in the PSA cryptoprocessor driver interface specification[Page 445, 1]. This specification defines the concept of PSA Crypto core, i.e. "An implementation of the PSA Cryptography API is composed of a core and zero or more drivers". The PSA specification also has the concept of storage locations for keys, through the type `psa_key_location_t`, which is used to access keys that don't have local storage[2]. This is leveraged mainly by opaque drivers mainly that use keys for which the key material is not readable by the PSA crypto core layer.

TF-M defines a software driver called `tfm_builtin_key_loader` that provides no acceleration but just defines a dedicated key location defined through the `TFM_BUILTIN_KEY_LOADER_KEY_LOCATION` define. By resorting to the entry points provided by this driver, the PSA Crypto core slot management subsystem can access keys stored in the underlying platform, validate key usage policies and allow PSA Crypto APIs to uniformly access builtin keys using the same call flows used with traditional local-storage keys.

This is implemented by hooking the entry points defined by the driver into the `library/psa_crypto_driver_wrappers.c` file provided by the PSA Crypto core. This is currently done manually but eventually could be just autogenerated by parsing a description of the driver entry points in the JSON format. These entry points are:

1. `tfm_builtin_key_loader_init`
2. `tfm_builtin_key_loader_get_key_buffer_size`
3. `tfm_builtin_key_loader_get_builtin_key`

The call flow for the entry points from the driver wrapper layer is as follows:

1. During the driver initialisation phase started by the PSA Crypto Core init, the `tfm_builtin_key_loader_init` function is called to probe the platform HAL and retrieve the builtin keys. Those keys are described by the types defined in the HAL layer interface header `tfm_plat_crypto_keys.h`. In particular global tables describing key properties and user policies must be implemented by each platform and are retrieved by the two accessor functions `tfm_plat_builtin_key_get_desc_table_ptr()` and `tfm_plat_builtin_key_get_policy_table_ptr()`. The keys are loaded from the platform in secure RAM in the TF-M Crypto partition with associated metadata. It's worth to note that the keys are loaded through callback functions which the platform lists in the key descriptor table.

2. Once the TF-M Crypto service is initialised, at runtime it might receive a request through an API call to use one of the builtin key IDs. Those IDs are described in the `tfm_builtin_key_ids.h` header. Platforms can override the default values providing their own header.

3. When the PSA Crypto core in the `psa_get_and_lock_key_slot()` function checks that the `key_id` being requested is in the builtin range region, it probes the platform through the function `mbedtls_psa_platform_get_builtin_key` which must be implemented by the TF-M Crypto service library abstraction layer. This function just checks the key descriptor table from the HAL to understand if such `key_id` is available in the platform and what are the corresponding `slot` and `lifetime` values associated to it. The lifetime contains the location of the key, which determines which driver is responsible for dealing with it (for the `tfm_builtin_key_loader` driver, that is `TFM_BUILTIN_KEY_LOADER_KEY_LOCATION`), while the `slot` value maps the key to the slot in the driver internal storage.

4. At this point, the PSA Crypto core knows that the key exists and is bound to the location associated to the driver. By calling into the driver wrappers layer is then able to retrieve the key attributes stored in the platform for that key ID, and the size required to allocate in its internal slot management system in order to load the key material in the core. This is done by calling `tfm_builtin_key_loader_get_builtin_key` just with a valid key attributes pointer (and nothing else), to retrieve the attributes. Once the attributes are available, the required size is retrieved through the driver wrapper by calling `tfm_builtin_key_loader_get_key_buffer_size`.

[2] Definition of psa_key_location_t type in the PSA spec: https://arm-software.github.io/psa-api/crypto/1.1/api/keys/lifetimes.html#c.psa_key_location_t

5. At this stage, the slot management subsystem calls again into the driver wrapper layer through `tfm_builtin_key_loader_get_builtin_key` with a valid buffer to hold the key material returned by the `tfm_builtin_key_loader` driver. When loading the key, the user requiring that key_id is validated by the driver code against the policies defined by the platform. If the policies match, the builtin key material and metadata is loaded and is used like a transparent key available to the PSA Crypto core slot management subsystem.

11.6.4 Technical details

Builtin key IDs and overriding

TF-M builtin key IDs are defined in `interface/include/crypto_keys/tfm_builtin_key_ids.h` through the enum `tfm_key_id_builtin_t`. They are allocated inside the range that PSA specifies for the builtin keys, i.e. between `MBEDTLS_PSA_KEY_ID_BUILTIN_MIN` and `MBEDLTS_PSA_KEY_ID_BUILTIN_MAX`. A platform can specify extra builtin key IDs by setting the `PLATFORM_DEFAULT_CRYPTO_KEYS` variable to `OFF`, creating the header `tfm_builtin_key_ids.h`, and specifying new keys and IDs.

Builtin key access control

A builtin key is accessible by all callers since the `key_id` associated to it is public information. Access to the keys must be mediated, which is done by matching the user requesting the `key_id` against the policies available for that user on that particular key in the policy table. If no policies are specified for a specific combination of user and `key_id`, the usage flags in the key attributes will be all set to zeros, meaning the key will be unusable for any operation for that particular user.

Multi-partition key derivation

The HUK is used for key derivation by any secure partition or NS caller that requires keys that are bound to a particular context. For example, Protected Storage derives keys uniquely for each user of the service which are used to encrypt user files. In order to provide HUK derivation to every secure partition / NS caller, it must be ensured that no service that utilises HUK derivation can derive the same key as another service (simply by using the same inputs for the KDF APIs, i.e. accessing the same base key for derivation).

This is accomplished by deriving a further "platform key" for each builtin key that has `PSA_KEY_USAGE_DERIVE` set in its attributes. These platform keys are derived from the builtin key, using the partition ID as a KDF input, and can then be used for safely for further derivations by the user, without risks to derive the same keys as other users. This is enforced directly by the `tfm_builtin_key_loader` driver.

Note: If the NS client ID feature is disabled, all NS callers share a partition ID of `-1`, and therefore will share a platform key and be therefore be able to derive the same keys as other NS callers.

For keys that are not exposed outside the device, this is transparent to the service that is using the key derivation, as they have no access to the builtin key material and cannot distinguish between keys derived directly from it and keys derived from the platform key. For some builtin keys, deriving platform keys is not acceptable, as the key is used outside the device (i.e. the IAK public key is used to verify attestation tokens) so the actual builtin key is used.

The decision has been taken to derive platform keys for any key that can be used for key derivation (`PSA_KEY_USAGE_DERIVE`), and not derive platform keys otherwise. For builtin keys that do not derive platform keys but are directly used, care must be taken with access control where multiple partitions have access to the same raw key material.

Mbed TLS transparent builtin keys

Mbed TLS does not natively support transparent builtin keys (transparent keys are keys where the key material is directly accessible by the PSA Crypto core), so some modifications had to be made. Opaque keyslots have the same basic structure as standard transparent key slots, and can be passed to the functions usually reserved for transparent keys, though this is a private implementation detail of the Mbed TLS library and is not specified in the driver interface. Therefore, the only modification required currently is to allow keys that have the location `TFM_BUILTIN_KEY_LOADER_KEY_LOCATION` to be passed to the functions that usually accept transparent keys only, i.e. with the location `PSA_KEY_LOCATION_LOCAL_STORAGE`. This is due to the fact that the standard assumption of the PSA Crypto core is that, if a driver that provides an additional location, will also provide dedicated cryptographic mechanisms to act on those keys, but this is not the case of the `tfm_builtin_key_loader`, as it just provides a mechanism to load keys (which act as a transparent key with local storage, once loaded), but Mbed TLS does not support such "transparent builtin key" concept. Note that the modifications on Mbed TLS are relying on non standard implementation details hence this particular integration can change between releases[3].

11.6.5 Using Opaque PSA crypto accelerators with TF-M

For platforms which have a cryptographic accelerator which has a corresponding Opaque PSA crypto accelerator driver, the TF-M builtin key loader driver can be disabled using the `-DCRYPTO_TFM_BUILTIN_KEYS_DRIVER=OFF` cmake option. The platform can then redefine the HAL function `tfm_plat_builtin_key_get_desc_table_ptr` to point to a table where the location and slot number of the keys corresponds instead to the opaque driver. The PSA driver wrapper will then route the calls into the opaque driver, with no other changes needed. If the description table is altered but the builtin key loader driver is not disabled, it is possible to mix software builtin keys with keys stored in opaque accelerators on a per-key level. Note that because the key policy enforcement via `tfm_plat_builtin_key_get_desc_table_ptr` is currently applied by the builtin key loader driver, other opaque drivers must apply either this policy or their own policy (Though this may be changed in future).

11.6.6 References

11.7 Log system design document

Author
Shawn Shan

Organization
Arm Limited

Contact
shawn.shan@arm.com

[3] Interface for platform keys: https://github.com/ARM-software/psa-crypto-api/issues/550

11.7.1 Background

In current TF-M log system, the SPM and Secure partitions share the same log APIs and implementations. While TF-M is keep evolving, the requirements for the log system has changed:

- Log level is required for both SPM and SP sides to output message in different scenarios.
- SPM only needs simple log format such as hex and string, while SP needs rich formatting.
- Distinctions on log output between SPM and SP are required.

A new log system is needed to separate the SPM and Secure partitions and to meet their different requirements.

11.7.2 Design

To allow customizable configurations, the log interfaces are defined as macros. The macros are easy to be forwarded or even empty. When SPM trying to output message and a value, it relies on a wrapper function, and finally output the formatted message by the HAL API.

The design principles of TF-M log system:

- Configurable log levels.
- Separated SPM and SP log implementations.
- Platforms provide log HAL implementations.

SPM Log System

Level Control

Three log levels for SPM log system are defined:

- TFM_SPM_LOG_LEVEL_DEBUG
- TFM_SPM_LOG_LEVEL_INFO
- TFM_SPM_LOG_LEVEL_ERROR
- TFM_SPM_LOG_LEVEL_SILENCE

Then a macro `TFM_SPM_LOG_LEVEL` is defined as an indicator, it should be equal to one of the four log levels.

API Definition

The following three APIs LOG APIs output the given 'msg' with hexadecimal formatted 'val' together. These APIs provide constrained ability to output numbers inside SPM. The 'msg' can be skipped with giving an empty string like "". And these APIs supports constant 'msg' string only, giving a runtime string as parameter 'msg' would potentially cause a runtime error.

> SPMLOG_DBGMSGVAL(msg, val);
>
> SPMLOG_INFMSGVAL(msg, val);
>
> SPMLOG_ERRMSGVAL(msg, val);

A C-function needs to work as an underlayer for these APIs as string formatting is required. Check 'spm_log_msgval' for details.

```
/**
 * brief Output the given message plus one value as hexadecimal. The message
 *       can be skipped if the 'msg' is 'NULL' or 'len' equals 0. The
 *       formatted hexadecimal string for 'value' has a '0x' prefix and
 *       leading zeros are not stripped. This function rely on HAL API
 *       'tfm_hal_output_spm_log' to output the formatted string.
 *
 * \param[in]  msg    A string message
 * \param[in]  len    The length of the message
 * \param[in]  value  A value need to be output
 *
 * \retval >=0        Number of chars output.
 * \retval <0         TFM HAL error code.
 */
int32_t  spm_log_msgval(const char *msg, size_t len, uint32_t value)
```

The following three APIs output a message in string.

SPMLOG_DBGMSG(msg);

SPMLOG_INFMSG(msg);

SPMLOG_ERRMSG(msg);

Here is a table about the effective APIs with different SPM log level.

	TFM_SPM_LOG_LEVEL_DEBUG	TFM_SPM_LOG_LEVEL_INFO	TFM_SPM_LOG_LEVEL_ERROR	TFM_SPM_LOG_LEVEL_SILENCE
SPMLOG_DBGMSGVAL	Yes	No	No	No
SPMLOG_INFMSGVAL	Yes	Yes	No	No
SPMLOG_ERRMSGVAL	Yes	Yes	Yes	No
SPMLOG_DBGMSG	Yes	No	No	No
SPMLOG_INFMSG	Yes	Yes	No	No
SPMLOG_ERRMSG	Yes	Yes	Yes	No

HAL API

Define HAL API for SPM log system:

```
/* SPM log HAL API */
int32_t tfm_hal_output_spm_log(const char *str, uint32_t len);
```

Take debug message as an example:

```
/* For debug message */
#define SPMLOG_DBGMSG(msg) tfm_hal_output_spm_log(msg, sizeof(msg))
/* For debug message with a value */
#define SPMLOG_DBGMSGVAL(msg, val) spm_log_msgval(msg, sizeof(msg), val)
```

Partition Log System

Partition log outputting required rich formatting in particular cases. There is a customized print inside TF-M(`printf`), and it is wrapped as macro.

Level Control

Three log levels for partition log system are defined:

- TFM_PARTITION_LOG_LEVEL_DEBUG
- TFM_PARTITION_LOG_LEVEL_INFO
- TFM_PARTITION_LOG_LEVEL_ERROR
- TFM_PARTITION_LOG_LEVEL_SILENCE

Then a macro `TFM_PARTITION_LOG_LEVEL` is defined as an indicator. It should be equal to one of the four log levels and it is an overall setting for all partitions.

Log Format

Compared to SPM, SP log API supports formatting. Similar to `printf`, these log APIs use a format outputting to output various type of data:

```
%d - decimal signed integer
%u - decimal unsigned integer
%x - hex(hexadecimal)
%c - char(character)
%s - string
```

API Definition

Define partition log APIs:

LOG_DBGFMT(. . .);

LOG_INFFMT(. . .);

LOG_ERRFMT(. . .);

Here is a table about the effective APIs with different partition log level.

	TFM_PARTITION_LOG_LEVEL_DEBUG	TFM_PARTITION_LOG_LEVEL_INFO	TFM_PARTITION_LOG_LEVEL_ERROR	TFM_PARTITION_LOG_LEVEL_SILEN
LOG_DBGFMT	Yes	No	No	No
LOG_INFFMT	Yes	Yes	No	No
LOG_ERRFMT	Yes	Yes	Yes	No

HAL API

Please refers to the HAL design document.

11.7.3 Log Devices

In most of the cases, a serial device could be used as a log device. And in other particular cases, a memory-based log device could be applied as well. These log device interfaces are abstracted into HAL APIs.

Note: It is not recommended to re-use the same HAL for both SPM and SP log outputting especially when SPM and SP run under different privileged level, which makes them have a different information confidential level. Unless:

- The SPM log outputting would be disabled as silence in the release version.

11.8 Physical attack mitigation in Trusted Firmware-M

Authors
Tamas Ban; David Hu

Organization
Arm Limited

Contact
tamas.ban@arm.com; david.hu@arm.com

11.8.1 Requirements

PSA Certified Level 3 Lightweight Protection Profile[1] requires protection against physical attacks. This includes protection against manipulation of the hardware and any data, undetected manipulation of memory contents, physical probing on the chip's surface. The RoT detects or prevents its operation outside the normal operating conditions (such as voltage, clock frequency, temperature, or external energy fields) where reliability and secure operation has not been proven or tested.

Note: Mitigation against certain level of physical attacks is a mandatory requirement for PSA Level 3 certification. The *TF-M countermeasures against physical attacks* discussed below doesn't provide mitigation against all the physical attacks considered in scope for PSA L3 certification. Please check the Protection Profile document for an exhaustive list of requirements.

[1] PSA Certified Level 3 Lightweight Protection Profile

11.8.2 Physical attacks

The goal of physical attacks is to alter the expected behavior of a circuit. This can be achieved by changing the device's normal operating conditions to untested operating conditions. As a result a hazard might be triggered on the circuit level, whose impact is unpredictable in advance but its effect can be observed. With frequent attempts, a weak point of the system could be identified and the attacker could gain access to the entire device. There is a wide variety of physical attacks, the following is not a comprehensive list rather just give a taste of the possibilities:

- Inject a glitch into the device power supply or clock line.
- Operate the device outside its temperature range: cool down or warm it up.
- Shoot the chip with an electromagnetic field. This can be done by passing current through a small coil close to the chip surface, no physical contact or modification of the PCB (soldering) is necessary.
- Point a laser beam on the chip surface. It could flip bits in memory or a register, but precise knowledge of chip layout and design is necessary.

The required equipment and cost of these attacks varies. There are commercial products to perform such attacks. Furthermore, they are shipped with a scripting environment, good documentation, and a lot of examples. In general, there is a ton of videos, research paper and blogs about fault injection attacks. As a result the threshold, that even non-proficient can successfully perform such attack, gets lower over time.

11.8.3 Effects of physical attacks in hardware and in software execution

The change in the behavior of the hardware and software cannot be seen in advance when performing a physical attack. On circuit-level they manifest in bit faults. These bit faults can cause varied effects in the behavior of the device micro-architecture:

- Instruction decoding pipeline is flushed.
- Altering instructions when decoding.
- Altering data when fetching or storing.
- Altering register content, and the program counter.
- Flip bits in register or memory.

These phenomenons happen at random and cannot be observed directly but the effect can be traced in software execution. On the software level the following can happen:

- A few instructions are skipped. This can lead to taking different branch than normal.
- Corrupted CPU register or data fetch could alter the result of a comparison instruction. Or change the value returned from a function.
- Corrupted data store could alter the config of peripherals.
- Very precise attacks with laser can flip bits in any register or in memory.

This is a complex domain. Faults are not well-understood. Different fault models exist but all of them target a specific aspect of fault injection. One of the most common and probably the easily applicable fault model is the instruction skip.

11.8.4 Mitigation against physical attacks

The applicability of these attacks highly depends on the device. Some devices are more sensitive than others. Protection is possible at hardware and software levels as well.

On the hardware level, there are chip design principles and system IPs that are resistant to fault injection attacks. These can make it harder to perform a successful attack and as a result the chip might reset or erase sensitive content. The device maker needs to consider what level of physical attack is in scope and choose a SoC accordingly.

On top of hardware-level protection, a secondary protection layer can be implemented in software. This approach is known as "defence in depth".

Neither hardware nor software level protection is perfect because both can be bypassed. The combination of them provides the maximum level of protection. However, even when both are in place, it is not certain that they provide 100% protection against physical attacks. The best of what is to achievable to harden the system to increase the cost of a successful attack (in terms of time and equipment), thereby making it non profitable to perform them.

Software countermeasures against physical attacks

There are practical coding techniques which can be applied to harden software against fault injection attacks. They significantly decrease the probability of a successful attack:

- Control flow monitor

  To catch malicious modification of the expected control flow. When an important portion of a program is executed, a flow monitor counter is incremented. The program moves to the next stage only if the accumulated flow monitor counter is equal to an expected value.

- Default failure

  The return value variable should always contain a value indicating failure. Changing its value to success is done only under one protected flow (preferably protected by double checks).

- Complex constant

  It is hard to change a memory region or register to a pre-defined value, but usual boolean values (0 or 1) are easier to manipulate.

- Redundant variables and condition checks

  To make branch condition attack harder it is recommended to check the relevant condition twice (it is better to have a random delay between the two comparisons).

- Random delay

  Successful fault injection attacks require very precise timing. Adding random delay to the code execution makes the timing of an attack much harder.

- Loop integrity check

  To avoid to skip critical loop iterations. It can weaken the cryptographic algorithms. After a loop has executed, check the loop counter whether it indeed has the expected value.

- Duplicated execution

  Execute a critical step multiple times to prevent fault injection from skipping the step. To mitigate multiple consecutive fault injections, random delay can be inserted between duplicated executions.

These techniques should be applied in a thoughtful way. If it is applied everywhere then it can result in messy code that makes the maintenance harder. Code must be analysed and sensitive parts and critical call path must be identified. Furthermore, these techniques increase the overall code size which might be an issue on the constrained devices.

Currently, compilers are not providing any support to implement these countermeasures automatically. On the contrary, they can eliminate the protection code during optimization. As a result, the C level protection does not add any guarantee about the final behavior of the system. The effectiveness of these protections highly depends on the actual compiler and the optimization level. The compiled assembly code must be visually inspected and tested to make sure that proper countermeasures are in-place and perform as expected.

11.8.5 TF-M Threat Model against physical attacks

Physical attack target

A malicious actor performs physical attack against TF-M to retrieve assets from device. These assets can be sensitive data, credentials, crypto keys. These assets are protected in TF-M by proper isolation.

For example, a malicious actor can perform the following attacks:

- Reopen the debug port or hinder the closure of it then connect to the device with a debugger and dump memory.
- Bypass secure boot to replace authentic firmware with a malicious image. Then arbitrary memory can be read.
- Assuming that secure boot cannot be bypassed then an attacker can try to hinder the setup of the memory isolation hardware by TF-M *Secure Partition Manager* (SPM) and manage to execute the non-secure image in secure state. If this is achieved then still an exploitable vulnerability is needed in the non-secure code which can be used to inject and execute arbitrary code to read the assets.
- Device might contain unsigned binary blob next to the official firmware. This can be any data, not necessarily code. If an attacker manages to replace this data with arbitrary content (e.g. a NOP slide leading to a malicious code) then they can try to manipulate the program counter to jump to this area before setting up the memory isolation.

Assumptions on attacker capability

It is assumed that the attacker owns the following capabilities to perform physical attack against devices protected by TF-M.

- Has physical access to the device.
- Able to access external memory, read and possibly tamper it.
- Able to load arbitrary candidate images for firmware upgrade.
- Able to manage that bootloader tries to upgrade the arbitrary image from staging area.
- Able to inject faults on hardware level (voltage or power glitch, EM pulse, etc.) to the system.
- Precise timing of fault injection is possible once or a few times, but in general the more intervention is required for a successful attack the harder will be to succeed.

It is out of the scope of TF-M mitigation if an attacker is able to directly tamper or disclose the assets. It is assumed that an attacker has the following technical limitations.

- No knowledge of the image signing key. Not able to sign an arbitrary image.
- Not able to directly access to the chip through debug port.
- Not able to directly access internal memory.
- No knowledge of the layout of the die or the memory arrangement of the secure code, so precise attack against specific registers or memory addresses are out of scope.

Physical attack scenarios against TF-M

Based on the analysis above, a malicious actor may perform physical attacks against critical operations in *SPE* workflow and critical modules in TF-M, to indirectly gain unauthenticated accesses to assets.

Those critical operations and modules either directly access the assets or protect the assets from disclosure. Those operations and modules can include:

- Image validation in bootloader
- Isolation management in TF-M, including platform specific configuration
- Cryptographic operations
- TF-M Secure Storage operations
- PSA client permission check in TF-M

The detailed scenarios are discussed in following sections.

Physical attacks against bootloader

Physical attacks may bypass secure image validation in bootloader and a malicious image can be installed.

The countermeasures is bootloader specific implementation and out of the scope of this document. TF-M relies on MCUboot by default. MCUboot has already implemented countermeasures against fault injection attacks[3].

Physical attacks against TF-M SPM

TF-M SPM initializes and manages the isolation configuration. It also performs permission check against secure service requests from PSA clients.

Static isolation configuration

It is TF-M SPM's responsibility to build up isolation during the initialization phase. If this is missed or not done correctly then it might be possible for non-secure code to access some secure memory area or an external device can access assets in the device through a debug port.

Therefore, hindering the setup of memory or peripheral isolation hardware is an obvious candidate for physical attacks. The initialization phase has a constant time execution (like the previous boot-up state), therefore the timing of the attack is simpler, compared to cases when secure and non-secure runtime firmware is up-and-running for a while and IRQs make timing unpredictable.

Some examples of attacking isolation configuration are shown in the list below.

- Hinder the setting of security regions. Try to execute non-secure code as secure.
- Manipulate the setting of secure regions, try to extend the non-secure regions to cover a memory area which otherwise is intended to be secure area.
- Hinder the setting of isolation boundary. In this case vulnerable ARoT code has access to all memory.
- Manipulate peripheral configuration to give access to non-secure code to a peripheral which is intended to be secure.

[3] MCUboot fault injection mitigation

PSA client permission checks

TF-M SPM performs several permission checks against secure service requests from a PSA client, such as:

- Check whether the PSA client is a non-secure client or a secure client

  NS client's PSA client ID is negative. NS client is not allowed to directly access secure areas. A malicious actor can inject faults when TF-M SPM authenticates a NS client. It may manipulate TF-M to accept it as a secure client and allow the NS client to access assets.

- Memory access checks

  TF-M SPM checks whether the request has correct permission to access a secure memory area. A malicious actor can inject faults when TF-M SPM checks memory access permission. It may skip critical check steps or corrupt the check result. Thereby a malicious service request may pass TF-M memory access check and accesses assets which it is not allowed to.

The physical attacks mentioned above relies on the a malicious NS application or a vulnerable RoT service to start a malicious secure service request to access the assets. The malicious actor has to be aware of the accurate timing of dealing with the malicious request in TF-M SPM. The timing can be affected by other clients and interrupts. It should be more difficult than pure fault injection.

Dynamic isolation boundary configuration

Physical attack may affect the isolation boundary setting during TF-M context switch, especially in Isolation Level 3. For example:

- A fault injection may cause TF-M SPM to skip clear privileged state before switching in an ARoT service.
- A fault injection may cause TF-M SPM to skip updating MPU regions and therefore the next RoT service may access assets belonging to a previous one.

However, it is much more difficult to find out the accurate timing of TF-M context switch, compared to other scenarios in TF-M SPM. It also requires a vulnerable RoT service to access assets after fault injection.

Physical attacks against TF-M Crypto service

Since crypto operations are done by mbedTLS library or by a custom crypto accelerator engine and its related software driver stack, the analysis of physical attacks against crypto operations is out-of-scope for this document. However, in general the same requirements are applicable for the crypto, to be compliant with PSA Level 3 certification. That is, it must be resistant against physical attacks. So crypto software and hardware must be hardened against side-channel and physical attacks.

Physical attacks against Secure Storage

Physical attacks against Internal Trusted Storage

Based on the assumption in *Assumptions on attacker capability*, a malicious actor is unable to directly retrieve assets via physical attacks against *Internal Trusted Storage* (ITS).

Instead, a malicious actor can inject faults into isolation configuration of ITS area in TF-M SPM to gain the access to assets stored in ITS. Refer to *Physical attacks against TF-M SPM* for details.

Physical attacks against Protected Storage

Based on the assumption in *Assumptions on attacker capability*, a malicious actor can be able to directly access external storage device. Therefore *Protected Storage* (PS) shall enable encryption and authentication by default to detect tampering with the content in external storage device.

A malicious actor can also inject faults into isolation configuration of PS and external storage device peripherals in TF-M SPM to gain the access to assets stored in PS. Refer to *Physical attacks against TF-M SPM* for details.

It is out of the scope of TF-M to fully prevent malicious actors from directly tampering with or retrieving content stored in external storage devices.

Physical attacks against platform specific implementation

Platform specific implementation includes critical TF-M HAL implementations. A malicious actor can perform physical attack against those platform specific implementations to bypass the countermeasures in TF-M common code.

Platform early initialization

TFM provides a HAL API for platforms to perform HW initialization before SPM initialization starts. The system integrator is responsible to implement this API on a particular SoC and harden it against physical attacks:

```
enum tfm_hal_status_t tfm_hal_platform_init(void);
```

The API can have several initializations on different modules. The system integrator can choose to even harden some of these initializations functions within this platform init API. One of the examples is the debug access setting.

Debug access setting

TF-M configures debug access according to device lifecycle and accessible debug certificates. In general, TF-M locks down the debug port if the device is in secure production state. The system integrator can put the settings into an API and harden it against physical attacks.

Platform specific isolation configuration

TFM SPM exposes a HAL API for static and dynamic isolation configuration. The system integrator is responsible to implement these API on a particular SoC and harden it against physical attacks.

```
enum tfm_hal_status_t tfm_hal_set_up_static_boundaries(void);
enum tfm_hal_status_t tfm_hal_bind_boundary(const struct partition_load_info_t *p_ldinf,
                                            uintptr_t *p_boundary);
```

Memory access check

TFM SPM exposes a HAL API for platform specific memory access check. The system integrator is responsible to implement this API on a particular SoC and harden it against physical attacks.

```
tfm_hal_status_t tfm_hal_memory_check(uintptr_t boundary,
                                      uintptr_t base,
                                      size_t size,
                                      uint32_t access_type);
```

11.8.6 TF-M countermeasures against physical attacks

This section propose a design of software countermeasures against physical attacks.

Fault injection hardening library

There is no open-source library which implements generic mitigation techniques listed in *Software countermeasures against physical attacks*. TF-M project implements a portion of these techniques. TF-M software countermeasures are implemented as a small library Fault Injection Hardening (FIH) in TF-M code base. A similar library was first introduced and tested in the MCUboot project (version 1.7.0)[2] which TF-M relies on.

The FIH library is put under TF-M `lib/fih/`.

The implementation of the different techniques was assigned to fault injection protection profiles. Four profiles (OFF, LOW, MEDIUM, HIGH) were introduced to fit better to the device capability (memory size, TRNG availability) and to protection requirements mandated by the device threat model. Fault injection protection profile is configurable at compile-time, default value: OFF.

Countermeasure profiles and corresponding techniques are listed in the table below.

Countermeasure	Profile LOW	Profile MEDIUM	Profile HIGH	Comments
Control flow monitor	Y	Y	Y	
Failure loop hardening	Y	Y	Y	
Complex constant		Y	Y	
Redundant variables and checks		Y	Y	
Random delay			Y	Implemented, but depends on HW capability

Similar to MCUboot, four profiles are supported. It can be configured at build time by setting (default is OFF):

```
-DTFM_FIH_PROFILE=<OFF, LOW, MEDIUM, HIGH>
```

[2] MCUboot project

How to use FIH library

As analyzed in *TF-M Threat Model against physical attacks*, this section focuses on integrating FIH library in TF-M SPM to mitigate physical attacks.

- Identify critical function call path which is mandatory for configuring isolation or debug access. Change their return types to `FIH_RET_TYPE` and make them return with `FIH_RET`. Then call them with `FIH_CALL`. These macros are providing the extra checking functionality (control flow monitor, redundant checks and variables, random delay, complex constant) according to the profile settings. More details about usage can be found here: `trusted-firmware-m/lib/fih/inc/fih.h`

  Take simplified TF-M SPM initialization flow as an example:

  ```
  main()
    |
    |--> tfm_core_init()
    |           |
    |           |--> tfm_hal_set_up_static_boundaries()
    |           |                  |
    |           |                  |--> platform specific isolation impl.
    |           |
    |           |--> tfm_hal_platform_init()
    |                              |
    |                              |--> platform specific init
    |
    |--> During each partition initialization
                |
                |--> tfm_hal_bind_boundary()
                                   |
                                   |--> platform specific peripheral isolation impl.
  ```

- Might make the important setting of peripheral config register redundant and verify them to match expectations before continue.

- Implements an extra verification function which checks the critical hardware config before secure code switches to non-secure. Proposed API for this purpose:

  ```
  fih_int tfm_hal_verify_static_boundaries(void);
  ```

  This function is intended to be called just after the static boundaries are set up and is responsible for checking all critical hardware configurations. The goal is to catch if something is missed and act according to system policy. The introduction of one more checking point requires one more intervention with precise timing. The system integrator is responsible to implement this API on a particular SoC and harden it against physical attacks. Make sure that all platform dependent security feature is properly configured.

- The most powerful mitigation technique is to add random delay to the code execution. This makes the timing of the attack much harder. However it requires an entropy source. It is recommended to use the `HIGH` profile when hardware support is available. There is a porting API layer to fetch random numbers in FIH library:

  ```
  void fih_delay_init(void);
  uint8_t fih_delay_random(void);
  ```

- Similar countermeasures can be implemented in critical steps in platform specific implementation.

  Take memory isolation settings on AN521 platform as an example. The following hardware components are responsible for memory isolation in a SoC, which is based on SSE-200 subsystem. System integrators must examine the chip specific memory isolation solution, identify the key components and harden the configuration of those. This list just serves as an example here for easier understanding:

- Implementation Defined Attribution Unit (IDAU): Implementation defined, it can be a static config or dynamic. Contains the default security access permissions of the memory map.
- SAU: The main module in the CPU to determine the security settings of the memory.
- *MPC*: External module from the CPU point of view. It protects the non security aware memories from unauthenticated access. Having a properly configured MPC significantly increases the security of the system.
- *PPC*: External module from the CPU point of view. Protects the non security aware peripherals from unauthenticated access.
- MPU: Protects memory from unprivileged access. ARoT code has only a restricted access in secure domain. It mitigates that a vulnerable or malicious ARoT partition can access to device assets.

The following AN521 specific isolation configuration functions shall be hardened against physical attacks.

```
sau_and_idau_cfg()
mpc_init_cfg()
ppc_init_cfg()
```

Some platform specific implementation rely on platform standard device driver libraries. It can become much more difficult to maintain drivers if the standard libraries are modified with FIH library. Platform specific implementation can implement duplicated execution and redundant variables/ condition check when calling platform standard device driver libraries according to usage scenarios.

Impact on memory footprint

The addition of protection code against physical attacks increases the memory footprint. The actual increase depends on the selected profile and where the mitigation code is added.

Attack experiment with SPM

The goal is to bypass the setting of memory isolation hardware with simulated instruction skips in fast model execution (FVP_MPS2_AEMv8M) in order to execute the regular non-secure test code in secure state. This is done by identifying the configuration steps which must be bypassed to make this happen. The instruction skip simulation is achieved by breakpoints and manual manipulation of the program counter. The following steps are done on AN521 target, but this can be different on another target:

- Bypass the configuration of isolation HW: SAU, MPC.
- Bypass the setting of the PSP limit register. Otherwise, a stack overflow exception will happen. Because the secure PSP will be overwritten by the address of the non-secure stack and on this particular target the non-secure stack is on lower address than the value in the secure PSP_LIMIT register.
- Avoid the clearing of the least significant bit in the non-secure entry point, where BLXNS/BXNS is jumping to non-secure code. Having the least significant bit cleared indicates to the hardware to switch security state.

The previous steps are enough to execute the non-secure Reset_Handler() in secure state. Usually, RTOS is executing on the non-secure side. In order to properly boot it up further steps are needed:

- Set the S_VTOR system register to point the address of the NS Vector table. Code is executed in secure state therefore when an IRQ hit then the handler address is fetched from the table pointed by S_VTOR register. RTOS usually do an SVC call at start-up. If S_VTOR is not modified then SPM's SVC handler will be executed.
- TBC: RTX osKernelStart still failing.

The bottom line is that in order to execute the regular non-secure code in secure state the attacker need to interfere with the execution flow at many places. Successful attack can be made even harder by adding the described mitigation techniques and some random delays.

11.8.7 Reference

CHAPTER

TWELVE

CONTRIBUTION GUIDELINES

12.1 Contributing Process

Contributions to the TF-M project need to follow the process below.

Note: Please contact TF-M mailing list for any question.

- It is recommended to subscribe to TF-M mailing list via this page.
- Refer to the *Roadmap* or send a mail to the TF-M mailing list to get the latest status and plan of TF-M.
- Follow *Design Proposal Guideline* to propose your design.
- Follow guidelines below to prepare the patch:
  - Clone the TF-M code on your own machine from TF-M git repository.
  - Follow the TF-M getting started, *Build Instructions Coding Guide* for the TF-M project.
  - Make your changes in logical chunks to help reviewers. Each commit should be a separate review and either work properly or be squashed after the review and before merging.
  - Follow *Documentation Contribution Guidelines* to update documentation in `docs` folder if needed.
  - Test your changes and add details to the commit description.
  - The code is accepted under *Developer Certificate of Origin (DCO)*. Use `git commit -s` to add a `Signed-off-by` trailer at the end of the commit log message. See git-commit for details.
  - Ensure that each changed file has the correct copyright and license information. Files that entirely consist of contributions to this project should have a copyright notice and BSD-3-Clause SPDX license identifier of the form as shown in *License*. Files that contain changes to imported Third Party IP files should retain their original copyright and license notices.

    Contributors can add the following copyright note, (whilst) it is suggested to update copyright note only for *major*, non-trivial changes.

    ```
    Copyright (c) XXXX[-YYYY], <OWNER>. All rights reserved.
    ```

    where XXXX is the year of first contribution and YYYY is the optional year of most recent contribution. <OWNER> is your or your company name.
  - Add a Change-Id to the commit message, which can be generated any way you like (e.g. from the SHA of the commit). It is suggested to clone repositories with commit-msg hook. The commit-msg hook attaches Change-Id automatically. Take trusted-firmware-m as an example.
- Submit your patch for review. Refer to Uploading Changes for details of uploading patch.

- Add relevant *code owner(s)* for reviewing the patch.
- You may be asked to provide further details or make additional changes.
- You can discuss further with code owner(s) and maintainer(s) directly via TF-M mailing list if necessary.
- If multiple patches are linked in a chain then code owners and maintainers should review and merge individual patches when they are ready, rather than waiting for the entire chain to be reviewed. If multiple patches are intended to be merged together then authors shall use topics or explicitly mention that in the commit message.
- Click `Allow-CI +1` button on Gerrit page to run CI to validate your patch. Your patch shall pass CI successfully before being merged. Code owner(s) and maintainer(s) may ask for additional test.
- Once the change is approved by code owners, the patch will be merged by the maintainer.

12.2 Code Review Guideline

The purpose of this document is to clarify design items to be reviewed during the code review process.

Please contact *maintainers* or write an e-mail thread on the TF-M mailing list for any questions.

12.2.1 List of the guidelines

The prerequisites before going to the review stage:

- Read the *Contributing Process* to know basic concepts.
- Read the *Source Structure* for structure related reference.

The review guidelines consist of these items:

- *Exceptions*.
- *Non-source items*.
- *Namespace*.
- *Assembler code*.
- *Include*.
- *Auto-doc*.
- *Commit Message*.

Note: Each guideline item is assigned with a unique symbol in the format `Rx.y`, while `x` and `y` are numeric symbols. These symbols are created for easy referencing during the code review process.

Exceptions

Files under listed folders are fully or partially imported from 3rd party projects, these files follow the original project defined guidelines and styles:

Important:

- R1.1 `ext` and its subfolders.

Non-source items

For all non-source files such as build system files or configuration files:

Important:

- R2.1 Follow the existing style while making changes.

Namespace

TF-M assign the namespace to files and symbols for an easier code reading. The symbol here includes functions, variables, types and MACROs. The prerequisites:

Important:

- R3.1 Follow the documents or specifications if they propose namespaces ('psa_' for PSA defined items E.g.,). Ask the contributor to tell the documents or specifications if the reviewer is not sure about a namespace.
- R3.2 Assign TF-M specific namespace 'tfm_' or 'TFM_' for symbols implementing TF-M specific features. Ask the contributor to clarify the purpose of the patch if the reviewer is not sure about the namespace.

For the sources out of the above prerequisites (R3.1 and R3.2):

Important:

- R3.3 Do not assign a namespace for source files.
- R3.4 Assigning a namespace for interface symbols is recommended. The namespace needs to be expressed in one word to be followed by other words into a phrase can represent the functionalities implemented. One word naming is allowed if the namespace can represent the functionality perfectly.
- R3.5 Assign a namespace for private symbols is NOT recommended.
- R3.6 Do not assign characters out of alphabets as the leading character.

Examples:

```
/* R3.1 FILE: s/spm/core/psa_client.c */

/* R3.2 FILE: s/spm/core/tfm_secure_context.c */

/* R3.4 FILE: s/spm/core/spm.c, 'spm\_' as the namespace */
```

(continues on next page)

(continued from previous page)

```
void spm_init(void);

/* R3.5 FILE: s/spm/core/main.c */
static void init_functions(void);

/* R3.6 Not permitted: */
/* static uint32_t __count; */
```

Assembler code

Important:

- R4.1 Pure assembler sources or inline assembler code are required to be put under the platform-independent or architecture-independent folders. The logic folders should not contain any assembler code, referring to external MACRO wrapped assembler code is allowed. Here is one example of the logic folder:
  - 'secure_fw/spm'.

Examples:

```
/*
 * R4.1 The following MACRO is allowed to be referenced under
 * 'secure_fw/spm'
 */
#define SVC(code) __asm volatile("svc  %0", ::"I"(code))
```

Include

This chapter describes the placement of the headers and including. There are two types of headers: The `interface` headers contain symbols to be shared between modules and the `private` headers contain symbols only for internal usage.

Important:

- R5.1 Put the `interface header` of one module in the `include` folder under the root of this module. Deeper sub-folders can not have `include` folders, which means only one `include` is allowed for one module.
- R5.2 Creating sub-folders under `include` to represent the more granular scope of the interfaces is allowed.
- R5.3 `private header` can be put at the same place with the implementation sources for the private symbols contained in the header. It also can be put at the place where the sources need it. The latter is useful when some "private header" contains abstracted interfaces, but these interfaces are not public interfaces so it won't be put under "include" folder.
- R5.4 Use <> when including public headers.
- R5.5 Use "" when including private headers.
- R5.6 The module's `include` folder needs to be added into referencing module's header searching path.
- R5.7 The module's `include` folder and the root folder needs to be added into its own header searching path and apply a hierarchy including with folder name.

- R5.8 Path hierarchy including is allowed since there are sub-folders under `include` folder and the module folder.
- R5.9 The including statement group order: the beginning group contains toolchain headers, then follows the public headers group and finally the private headers group.
- R5.10 The including statement order inside a group: Compare the include statement as strings and sort by the string comparison result.
- R5.11 The header for the referenced symbol or definition must be included even this header is included inside one of the existing included headers. This improves portability in case the existing header implementation changed.

Examples:

```
/*
 * The structure:
 *   module_a/include/func1.h
 *   module_a/include/func2/fmain.h
 *   module_a/func1.c
 *   module_a/func2/fmain.c
 *   module_b/include/funcx.h
 *   module_b/include/funcy/fmain.h
 *   module_b/funcx.c
 *   module_b/funcxi.h
 *   module_b/funcy/fmain.c
 *   module_b/funcy/fsub.c
 *   module_b/funcy/fsub.h
 * Here takes module_b/funcx.c as example:
 */
#include <func1.h>
#include <func2/fmain.h>
#include <funcx.h>
#include "funcxi.h"
#include "funcy/fsub.h"
```

Auto-doc

Auto document system such as doxygen is useful for describing interfaces. While it would be a development burden since the details are described in the design documents already. The guidelines for auto-doc:

Important:

- R6.1 Headers and sources under these folders need to apply auto-doc style comments: `*include`.
- R6.2 Developers decide the comment style for sources out of listed folders.

Commit Message

TF-M has the requirements on commit message:

Important:

- R7.1 Assign correct topic for a patch. Check the following table.

Topic	Justification
Boot	*bl2/** or *bl1/**
Build	For build system related purpose.
Docs	All *.rst changes.
Dualcpu	Dual-cpu related changes.
HAL	Generic HAL interface/implementation changes.
Interface	Interface changes, either Non-source and secure.
Pack	For packing purpose.
Platform	Generic platform related changes under *platform/**.
Platform Name	Specific platform changes.
Partition	Multiple partition related changes.
Partition Name	Specific partition related changes.
Service	Multiple service related changes.
Service Name	Specific service related changes.
SPM	*secure_fw/spm/**
SPRTL	*secure-fw/partitions/lib/runtime/**
Tool	*tools* folder or *tf-m-tools* repo

Note: Ideally, one topic should cover one specific type of changes. For crossing topic changes, check the main part of the change and use the main part related topic as patch topic. If there is no suitable topics to cover the change, contact the community for an update.

Trusted Firmware-M (TF-M) is an open govcommunity project. All contributions are ultimately merged by the maintainers listed below. Technical ownership of most parts of the codebase falls on the code owners listed below. An acknowledgement from these code owners is required before the maintainers merge a contribution.

More details may be found in the Project Maintenance Process document.

12.3 Maintainers

Anton Komlev

email
Anton.Komlev@arm.com

github
Anton-TF

Antonio de Angelis

email
Antonio.deAngelis@arm.com

github
adeaarm

Chris Brand

email
Chris.Brand@cypress.com

github
UEWBot

David Hu

email
David.Hu@arm.com

github
davidhuziji

Ken Liu

email
Ken.Liu@arm.com

github
KenLSoft

12.4 Code owners

12.4.1 Bootloader and FWU

Tamas Ban

email
Tamas.Ban@arm.com

github
tamasban

Sherry Zhang

email
Sherry.Zhang2@arm.com

github
sherryzhang

12.4.2 BL1 immutable bootloader

Raef Coles

email
Raef.Coles@arm.com

github
RcColes

12.4.3 Secure Storage

Jamie Fox

email
jamie.fox@arm.com

github
jf549

Sherry Zhang

email
Sherry.Zhang2@arm.com

github
sherryzhang

12.4.4 Crypto

Antonio de Angelis

email
Antonio.deAngelis@arm.com

github
adeaarm

Summer Qin

email
Summer.Qin@arm.com

github
Summer-ARM

12.4.5 Framework (SPM, etc.)

Ken Liu

email
Ken.Liu@arm.com

github
KenLSoft

Kevin Peng

email
Kevin.Peng@arm.com

github
kevin-peng-hao

12.4.6 Attestation

Maulik Patel

email
Maulik.Patel@arm.com

github
maulik-arm

David Hu

email
David.Hu@arm.com

github
davidhuziji

12.4.7 Build System

Raef Coles

email
Raef.Coles@arm.com

github
RcColes

Anton Komlev

email
Anton.Komlev@arm.com

github
Anton-TF

12.4.8 Tests

David Hu

email
David.Hu@arm.com

github
davidhuziji

Kevin Peng

email
Kevin.Peng@arm.com

github
kevin-peng-hao

12.4.9 Arm Platforms

MPS2, MPS3, MPS4, Musca(B1,S1)

David Hazi

email
David.Hazi@arm.com

github
david-hazi-arm

Corstone1000

Xueliang Zhong

email
Xueliang.Zhong@arm.com

github
xueliang-zhong

Emekcan Aras

email
Emekcan.Aras@arm.com

github
ememarar

RSE

Jamie Fox

email
jamie.fox@arm.com

github
jf549

12.4.10 NXP Platforms

Andrej Butok

email
Andrey.Butok@nxp.com

github
butok

12.4.11 STM Platforms: DISCO_L562QE, NUCLEO_L552ZE_Q

Michel JAOUEN

email
Michel.Jaouen@st.com

github
jamike

12.4.12 Infineon/Cypress Platforms

Chris Brand

email
Chris Brand@cypress.com

github
UEWBot

12.4.13 Laird Connectivity Platforms:

Greg Leach

email
Greg.Leach@lairdconnect.com

github
greg-leach

12.4.14 Nordic Semiconductor Platforms

Georgios Vasilakis

email
georgios.vasilakis@nordicsemi.no

github
Vge0rge

12.4.15 Nuvoton Platform:

WS Chang

email
MS20 WSChang0@nuvoton.com

github
wschang0

12.4.16 ArmChina Platform:

Jidong Mei

email
Jidong.Mei@armchina.com

github
JidongMei

12.5 Yet another coding standard :)

Warning: Every rule has an exception so if you disagree or dislike then reach out!

The coding style of TF-M project is based on Linux coding style but there are updates for domain specific conventions as listed below.

TF-M also reuses code from other SW projects, e.g. `CMSIS_5`, which means some areas of code may have different styles. We use common sense approach and new code may inherit coding style from external projects but it needs to remain within clear scope.

The guidance below is provided as a help. It isn't meant to be a definitive list.

As implied in the *contributing guide* maintainers have the right to decide on what's acceptable in case of any divergence.

Warning: Text files do not fall within these rules as they may require different formatting.

12.5.1 Consistent style

The code needs to be consistent with itself, so if existing code in the file violates listed coding style rules then it is better to follow existing style in the file and not break consistency by following the rules listed here.

You may need to add a comment in the commit description to clarify this.

List of rules

- Use 4 spaces indentation. No tabs.
- Use `lower_case_with_underscore` for filenames, variable and function names.
- Use standard types e.g. `uint32_t`, `uint16_t`, `uint8_t`, `int32_t` etc.
- Use `uintptr_t` type when representing addresses as numbers.
- Use static for all private functions and variables.
- Use void argument if your function doesn't contain any argument.
- Use C90 `/* */` for comments rather than C99 `//`
- No trailing spaces in code.

- Keep up to 100 characters per line but 140 chars can be accepted exceptionally.
- Open curly brace `{` at the same if/else/while/for/switch statement line.
- Use curly braces `{ }` for one line statement bodies also.
- Put open curly brace in the line below the function header.
- Order function parameters so that input params are before output params.
- Declare local variables at the beginning of the function.
- Define macros in all caps i.e. `CAPITAL_WITH_UNDERSCORE`.
- Use typedefs **ONLY** for function prototype declarations.
- Type definitions in `lower_case_with_underscore` ended by `_t`.
- Do not use typedef for other constructs as it obfuscates code.
- Do not use bitfields.
- Use static for all global variables to reduce the variable scope.
- Use enumeration for error codes to keep the code readable.
- Use descriptive variable and functions names.
- Put the correct license header at the beginning of the file.
- Keep the files (.h/.c) self-contained, i.e. put include dependencies in the file.
- Put appropriate header define guard in .h files: `__HEADER_NAME__`. Any divergence from this rules should be clearly documented.
- In a .c file, #include it's own header file first.
- Document all the public functions in the header file only.
- Document all the private functions in the source file only.
- Add "extern C " definition for C++ support in the header files.
- In the switch statement, aligned cases with the switch keyword.
- For enums, use upper case letters with digits and underscore only.
- Do not code while eating.

12.6 Documentation Contribution Guidelines

The Trusted Firmware-M project uses Sphinx to generate the Official Documentation from Restructed Text *.rst* source files,

The aim is to align as much as possible with the official Python Documentation Guidelines while keeping the consistency with the already existing files.

The guidance below is provided as a help. It is not meant to be a definitive list.

12.6.1 Overview

The following short-list provides a quick summary of the rules.

- If the patch modifies a present file, the file's style should be followed
- If creating a new file, *integration guide* can be used as a reference.
- When a new style is to be expressed, consult the Python Documentation Guidelines

12.6.2 List of rules

The following rules should be considered:

1. H1 document title headers should be expressed by # with over-line (rows on top and bottom) of the text
2. Only ONE H1 header should allowed per document, ideally placed on top of the page.
3. H2 headers should be expressed by * with over-line
4. H2 header's text should be UNIQUE in per document basis
5. H3 headers should be expressed by an underline of '='
6. H4 headers should be expressed by an underline of '-'
7. H3 and H4 headers have no limitation about naming. They can have similar names on the same document, as long as they have different parents.
8. H5 headers should be expressed by an underline of '^'
9. H5 headers will be rendered in document body but not in menus on the navigation bar
10. Do not use more than 5 levels of heading
11. When writing guides, which are expected to be able to be readable by command line tools, it would be best practice to add long complicated tables, and UML diagrams in the bottom of the page and using internal references(auto-label)
12. No special formatting should be allowed in Headers (code, underline, strike-through etc)
13. Long URLs and external references should be placed at the bottom of the page and referenced in the body of the document
14. New introduced terms and abbreviations should be added to Glossary and directly linked by the *:term:* notation across all documents using it.

12.6.3 Platform Documentation

The Documentation Build system provides an interface with the platform directory allowing maintainers to bundle platform specific documentation. Platforms are grouped by vendor. **This behaviour needs to be explicitly enabled for each vendor's space** by providing the *<vendor>/index.rst* (responsible for generating the *Platform Index File*) and adding a table of contents entry for the corresponding vendor's space. The format and structure of this entry is not strictly defined, and allows flexible control of vendor's and platform's documentation space. Follow the *Platform Documentation* document for more details.

12.6.4 Common Use Cases

The section below describes with examples, a rule compliant implementation for most common documentation elements.

Headers

```
###################
Document Title (H1)
###################

******************
Chapter Title (H2)
******************

Chapter Section (H3)
====================

Chapter Sub-Section (H4)
------------------------

Subsection of Chapter sub-section (H5)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Code Blocks

The recommendation for code content, is to use the explicit code-block directive, ideally with a defined lexer for the language the block contains.

A list of compatible lexers can be found at Pygments Lexers

```
.. code-block:: bash

    ls
    pwd

.. code-block:: doscon

    dir

.. code-block:: c

    static struct rn_object_t;

.. code-block:: python3

    print("Hello TF-M")
```

Restructured Text supports implicit code-blocks by indenting a section of text, surrounded by new lines. While this formatting is allowed, it should be avoided if possible.

```
The quick brown fox jumps over the lazy dog

    ls
    pwd
```

Note: Mixing two different code-block formats in the same document will break the whole document's rendering. When editing an existing document, please follow the existing format.

New documents should always use the explicit format.

Tables

For adding new tables the *table::* notation should be used.

```
.. table:: Components table
   :widths: auto

   +--------------+--------------+-------------+
   | **Title A**  | **Title B**  | **Title C** |
   +==============+==============+=============+
   | Text A       | Text B       | Text C      |
   +--------------+--------------+-------------+
```

While the equivalent simple table code will render correctly in the output, it will not be added to the index (So it cannot be referenced if needed)

```
+--------------+--------------+-------------+
| **Title A**  | **Title B**  | **Title C** |
+==============+==============+=============+
| Text A       | Text B       | Text C      |
+--------------+--------------+-------------+
```

Other types of tables such as list-tables and csv-tables are also permitted, as seen on /getting_started/tfm_getting_started and /releases/1.0

External Links

External links should be placed in the bottom of a document.

```
The quick brown fox jumps over the lazy dog according to `Link_A`_

.. _Link_A: https://www.aaa.org
.. _Link_B: https://www.bbb.org

--------------

*Copyright (c) XYZ *
```

Creating in-line links is permitted, but should be avoided if possible. It should be only used for consistency purposes or for a small ammount of short links.

```
The quick brown fox jumps over the lazy dog according to `Link_A <https://www.aaa.org>`_
```

If for the purposes of content, a link is to be referenced by multiple labels, internal linking is the recommended approach.

```
The quick brown fox jumps over the lazy dog according to `Link_A_New`_

.. _Link_A: https://www.aaa.org
.. _Link_A_New: `Link_A`_

--------------

*Copyright (c) XYZ *
```

Document Links

A document included in the documentation can be referenced by the *doc:* notation

```
:doc:`integration guide </integration_guide/tfm_integration_guide>`
```

The path is relative to the root of the Trusted Firmware-M code.

Trusted Firmware-M project is spread among multiple repositories: Trusted Firmware-M, TF-M Tests, TF-M Tools and TF-M Extras. Every repository has its own documentation, and since *v2.0.0*, they can be found under *Links*.

Using Intersphinx, it is now possible to use cross-reference roles from Sphinx to reference documentation from different projects (repositories), like TF-M Tests. Referencing documentation using the *doc:* notation is preferred and helps to avoid broken cross-references if the link of the document changes.

For example, to get this: Adding TF-M Regression Test Suite; the Restructed Text would look like this:

```
:doc:`Adding TF-M Regression Test Suite <TF-M-Tests:tfm_test_suites_addition>`
```

As can be seen, it is quite similar to cross-referencing in Sphinx, except the path to the document is preceded with the external project name. For TF-M Tests, it is `TF-M-Tests`. The names of other projects configured to be referenced using Intersphinx can be seen in the conf.py file under `intersphinx_mapping`.

Reference specific section of a document

In order to reference a specific section of a document, up to level 4 headers (if they are included in the index), the *ref:* keyword can be used

```
:ref:`docs/getting_started/tfm_getting_started:Tool & Dependency overview`
```

This can also be used to quickly scroll to the specific section of the current document. This technique can be used to add complex table in the bottom of a document and create clickable quick access references to it for improved user experience.

Glossary term

For technical terms and abbreviations, the recommended guidance is to add an entry to the *Glossary of terms and abbreviations* and refer to it, using the *term:* directive

```
HAL
Hardware Abstraction Layer
    Interface to abstract hardware-oriented operations and provides a set of
    APIs to the upper layers.

.....

As described in the design document :term:`HAL` abstracts the
hardware-oriented and platform specific
.......
```

Note: The ":term:" directive does not work when used in special formatting. Using *:term:_HAL_* **will not link to the glossary term**.

References

1. Sphinx
2. Restructed Text
3. Python Documentation Guidelines
4. Pygments Lexers
5. Intersphinx

12.7 Design proposal guideline

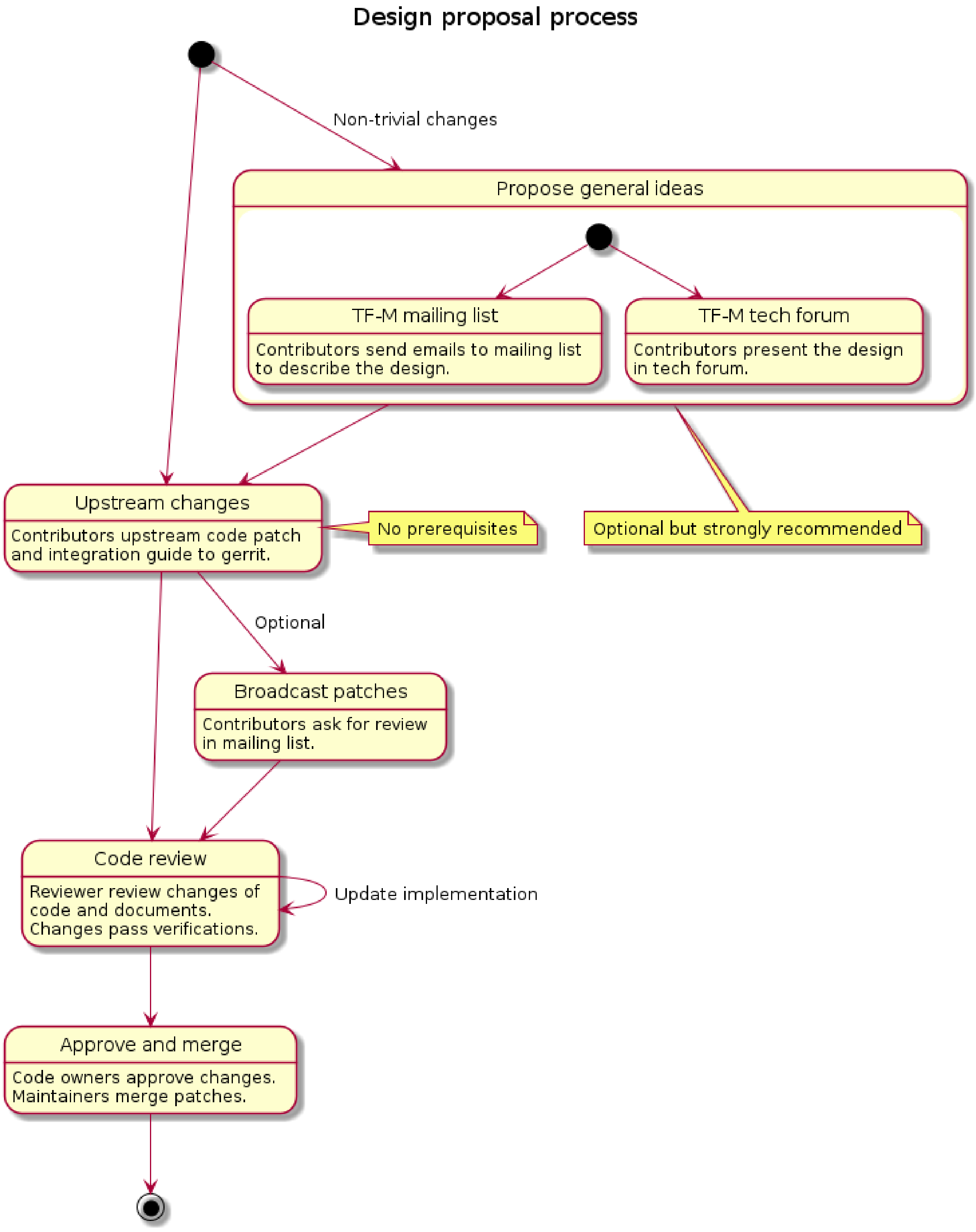
The design proposal guideline specifies the steps to propose and upload design proposals to TF-M. Those steps are lightweight and flexible to make sure that contributors can focus more on actual code implementation and iteration.

The guideline encourages developers to share design proposal via TF-M mailing list[1] and TF-M technical forum (tech forum)[2]. The design details can be discussed via code reviews of actual implementations.

Typical steps are shown as the diagram below.

[1] TF-M mailing list

[2] TF-M technical forum


Design proposal process
Non-trivial changes
Propose general ideas
TF-M mailing list
Contributors send emails to mailing list to describe the design.
TF-M tech forum
Contributors present the design in tech forum.
Upstream changes
Contributors upstream code patch and integration guide to gerrit.
No prerequisites
Optional but strongly recommended
Optional
Broadcast patches
Contributors ask for review in mailing list.
Code review
Reviewer review changes of code and documents. Changes pass verifications.
Update implementation
Approve and merge
Code owners approve changes. Maintainers merge patches.

12.7.1 Discussion in TF-M mailing list and tech forum

It is **highly recommended** to propose and discuss designs in TF-M mailing list or TF-M tech forum, before or while the code implementation is under review.

It is efficient and flexible to directly discuss design proposal via TF-M mailing list and TF-M tech forum. Contributors can receive quick and broad feedback from TF-M community.

Although it is optional to present the ideas in mailing list or tech forum, it will help reviewers understand the design much better and expedite the code review process.

12.7.2 Code review of details

It is straightforward and convenient for contributors and reviewers to deliberate over design and implementation details via code review.

Contributors can implement their design proposal and upstream the patch set to TF-M gerrit[3] for code review. For non-trivial changes or new major features, it is **strongly suggested** to propose the design to TF-M mailing list and tech forum in advance.

Contributors don't have to wait for any approvals before upstreaming patches, even if the changes are non-trivial. No formal design document in advance is required anymore.

The review process is the same as the general one[4], with some specific requirements:

- Contributors can send an email to TF-M mailing list to ask for review.
- If it requires additional reviewers besides code owners and maintainers, contributors shall add the specific reviewers in the review list.
- Authors shall clearly specify the design purpose and briefly describe the implementation in the commit message.
- Authors shall put essential comments and notes in code for the code changes.

Code owners and maintainers may require contributors to further verify the implementation besides normal per-patch CI test. Contributors shall provide the verification results as requested.

12.7.3 Integration guide and manual

Contributors can create an integration guide or a user manual to describe how to integrate the new features related to the design proposal.

Contributors shall update the corresponding documents if the design changes existing implementation.

Reference

[3] TF-M gerrit

[4] *Contributing process*

CHAPTER THIRTEEN

LICENSE

—

Note: Individual files contain the following tag instead of the full license text.

SPDX-License-Identifier: BSD-3-Clause

This enables machine processing of license information based on the SPDX License Identifiers that are here available: http://spdx.org/licenses/

CHAPTER

FOURTEEN

DEVELOPER CERTIFICATE OF ORIGIN

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

BIBLIOGRAPHY

[Security-Incident-Process] Security Incident Process

[FF-M] Arm® Platform Security Architecture Firmware Framework 1.0

[DUAL-CPU-BOOT] *Booting a dual core system*

[CVSS] Common Vulnerability Scoring System Version 3.1 Calculator

[CVSS_SPEC] CVSS v3.1 Specification Document

[STRIDE] The STRIDE Threat Model

[SECURE-BOOT] *Secure boot*

[ROLLBACK-PROTECT] *Rollback protection in TF-M secure boot*

[STACK-SEAL] Armv8-M processor Secure software Stack Sealing vulnerability

[ADVISORY-TFMV-1] *Advisory TFMV-1*

[ADVISORY-TFMV-2] *Advisory TFMV-2*

[PSA-FF-M] Arm Platform Security Architecture Firmware Framework 1.0

[RFC7925] Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things

[PROFILE-S] *Trusted Firmware-M Profile Small Design*

[RFC7252] The Constrained Application Protocol (CoAP)

[RFC4279] Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)

[RFC7251] AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS

[CRYPTO-DESIGN] *Crypto design*

[ITS-INTEGRATE] *ITS integration guide*

[TFM-BUILD] *TF-M build instruction*

[a] PS service is enabled by default. Platforms without off-chip storage devices can turn off `TFM_PARTITION_PROTECTED_STORAGE` to disable PS service. See *Protected Storage Secure Partition* for details.

[TBSA-M] Arm Platform Security Architecture Trusted Base System Architecture for Armv6-M, Armv7-M and Armv8-M, version 1.0

[HKDF] Hugo Krawczyk. 2010. Cryptographic extraction and key derivation: the HKDF scheme. In Proceedings of the 30th annual conference on Advances in cryptology (CRYPTO'10)

[RFC5869] IETF RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

INDEX