Trusted Firmware-A

unknown

Nov 17, 2020

CONTENTS

CHAPTER

ONE

ABOUT

1.1 Feature Overview

This page provides an overview of the current *TF-A* feature set. For a full description of these features and their implementation details, please see the documents that are part of the *Components* and *System Design* chapters.

The *Change Log & Release Notes* provides details of changes made since the last release.

1.1.1 Current features

- Initialization of the secure world, for example exception vectors, control registers and interrupts for the platform.
- Library support for CPU specific reset and power down sequences. This includes support for errata workarounds and the latest Arm DynamIQ CPUs.
- Drivers to enable standard initialization of Arm System IP, for example Generic Interrupt Controller (GIC), Cache Coherent Interconnect (CCI), Cache Coherent Network (CCN), Network Interconnect (NIC) and TrustZone Controller (TZC).
- A generic *SCMI* driver to interface with conforming power controllers, for example the Arm System Control Processor (SCP).
- SMC (Secure Monitor Call) handling, conforming to the SMC Calling Convention using an EL3 runtime services framework.
- *PSCI* library support for CPU, cluster and system power management use-cases. This library is pre-integrated with the AArch64 EL3 Runtime Software, and is also suitable for integration with other AArch32 EL3 Runtime Software, for example an AArch32 Secure OS.
- A minimal AArch32 Secure Payload (*SP_MIN*) to demonstrate *PSCI* library integration with AArch32 EL3 Runtime Software.
- Secure Monitor library code such as world switching, EL1 context management and interrupt routing. When a Secure-EL1 Payload (SP) is present, for example a Secure OS, the AArch64 EL3 Runtime Software must be integrated with a Secure Payload Dispatcher (SPD) component to customize the interaction with the SP.
- A Test SP and SPD to demonstrate AArch64 Secure Monitor functionality and SP interaction with PSCI.
- SPDs for the OP-TEE Secure OS, NVIDIA Trusted Little Kernel and Trusty Secure OS.
- A Trusted Board Boot implementation, conforming to all mandatory TBBR requirements. This includes image authentication, Firmware Update (or recovery mode), and packaging of the various firmware images into a Firmware Image Package (FIP).
- Pre-integration of TBB with the Arm CryptoCell product, to take advantage of its hardware Root of Trust and crypto acceleration services.

- Reliability, Availability, and Serviceability (RAS) functionality, including
  - A Secure Partition Manager (SPM) to manage Secure Partitions in Secure-EL0, which can be used to implement simple management and security services.
  - An *SDEI* dispatcher to route interrupt-based *SDEI* events.
  - An Exception Handling Framework (EHF) that allows dispatching of EL3 interrupts to their registered handlers, to facilitate firmware-first error handling.
- A dynamic configuration framework that enables each of the firmware images to be configured at runtime if required by the platform. It also enables loading of a hardware configuration (for example, a kernel device tree) as part of the FIP, to be passed through the firmware stages. This feature is now incorporated inside the firmware configuration framework (fconf), which is still flagged as experimental.
- Support for alternative boot flows, for example to support platforms where the EL3 Runtime Software is loaded using other firmware or a separate secure system processor, or where a non-TF-A ROM expects BL2 to be loaded at EL3.
- Support for the GCC, LLVM and Arm Compiler 6 toolchains.
- Support for combining several libraries into a “romlib” image that may be shared across images to reduce memory footprint. The romlib image is stored in ROM but is accessed through a jump-table that may be stored in read-write memory, allowing for the library code to be patched.
- Support for the Secure Partition Manager Dispatcher (SPMD) component as a new standard service.
- Support for ARMv8.3 pointer authentication in the normal and secure worlds. The use of pointer authentication in the normal world is enabled whenever architectural support is available, without the need for additional build flags. Use of pointer authentication in the secure world remains an experimental configuration at this time and requires the `BRANCH_PROTECTION` option to be set to non-zero.
- Position-Independent Executable (PIE) support. Currently for BL2, BL31, and TSP, with further support to be added in a future release.

1.1.2 Still to come

- Support for additional platforms.
- Refinements to Position Independent Executable (PIE) support.
- Continued support for the PSA FF-A v1.0 (formally known as SPCI) specification, to enable the use of secure partition management in the secure world.
- Documentation enhancements.
- Ongoing support for new architectural features, CPUs and System IP.
- Ongoing support for new Arm system architecture specifications.
- Ongoing security hardening, optimization and quality improvements.

1.2 Release Processes

1.2.1 Project Release Cadence

The project currently aims to do a release once every 6 months which will be tagged on the master branch. There will be a code freeze (stop merging non-essential changes) up to 4 weeks prior to the target release date. The release candidates will start appearing after this and only bug fixes or updates required for the release will be merged. The maintainers are free to use their judgement on what changes are essential for the release. A release branch may be created after code freeze if there are significant changes that need merging onto the integration branch during the merge window.

The release testing will be performed on release candidates and depending on issues found, additional release candidates may be created to fix the issues.

```
                    |<----------6 months---------->|
     |<---4 weeks--->|              |<---4 weeks--->|
+-----------------------------------------------------------> time
     |               |              |               |
  code freeze       ver w.x      code freeze     ver y.z
```

Upcoming Releases

These are the estimated dates for the upcoming release. These may change depending on project requirement and partner feedback.

Release Version	Target Date	Expected Code Freeze
v2.0	1st week of Oct '18	1st week of Sep '18
v2.1	5th week of Mar '19	1st week of Mar '19
v2.2	4th week of Oct '19	1st week of Oct '19
v2.3	4th week of Apr '20	1st week of Apr '20
v2.4	2nd week of Nov '20	4th week of Oct '20
v2.5	2nd week of May '21	4th week of Apr '21

1.2.2 Removal of Deprecated Interfaces

As mentioned in the *Platform Compatibility Policy*, this is a live document cataloging all the deprecated interfaces in TF-A project and the Release version after which it will be removed.

Interface	Deprecation Date	Removed after Release	Comments

1.3 Project Maintenance

Trusted Firmware-A (TF-A) is an open governance community 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.

1.3.1 Maintainers

M Dan Handley <dan.handley@arm.com>

G danh-arm

M Soby Mathew <soby.mathew@arm.com>

G soby-mathew

M Sandrine Bailleux <sandrine.bailleux@arm.com>

G sandrine-bailleux-arm

M Alexei Fedorov <Alexei.Fedorov@arm.com>

G AlexeiFedorov

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

M Mark Dykes <mark.dykes@arm.com>

G mardyk01

M Olivier Deprez <olivier.deprez@arm.com>

G odeprez

M Bipin Ravi <bipin.ravi@arm.com>

G bipinravi-arm

M Joanna Farley <joanna.farley@arm.com>

G joannafarley-arm

M Julius Werner <jwerner@chromium.org>

G jwerner-chromium

M Varun Wadekar <vwadekar@nvidia.com>

G vwadekar

M Andre Przywara <andre.przywara@arm.com>

G Andre-ARM

M Lauren Wehrmeister <Lauren.Wehrmeister@arm.com>

G laurenw-arm

M Madhukar Pappireddy <Madhukar.Pappireddy@arm.com>

G madhukar-Arm

M Raghu Krishnamurthy <raghu.ncstate@icloud.com>

G raghuncstate

1.3.2 Code owners

Core Code

Armv7-A architecture port

M Etienne Carriere <etienne.carriere@linaro.org>

G etienne-lms

Build Definitions for CMake Build System

M Javier Almansa Sobrino <Javier.AlmansaSobrino@arm.com>

G javieralso-arm

M Chris Kay <chris.kay@arm.com>

G CJkay

F /

Software Delegated Exception Interface (SDEI)

M Mark Dykes <mark.dykes@arm.com>

G mardyk01

M John Powell <John.Powell@arm.com>

G john-powell-arm

F services/std_svc/sdei/

Trusted Boot

M Sandrine Bailleux <sandrine.bailleux@arm.com>

G sandrine-bailleux-arm

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

M Manish Badarkhe <manish.badarkhe@arm.com>

G ManishVB-Arm

F drivers/auth/

Secure Partition Manager (SPM)

M Olivier Deprez <olivier.deprez@arm.com>

G odeprez

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

M Maksims Svecovs <maksims.svecovs@arm.com>

G max-shvetsov

M Joao Alves <Joao.Alves@arm.com>

G J-Alves

F services/std_svc/spm*

Exception Handling Framework (EHF)

M Manish Badarkhe <manish.badarkhe@arm.com>

G ManishVB-Arm

M John Powell <John.Powell@arm.com>

G john-powell-arm

F bl31/ehf.c

Drivers, Libraries and Framework Code

Console API framework

M Julius Werner <jwerner@chromium.org>

G jwerner-chromium

F drivers/console/

F include/drivers/console.h

F plat/common/aarch64/crash_console_helpers.S

coreboot support libraries

M Julius Werner <jwerner@chromium.org>

G jwerner-chromium

F drivers/coreboot/

F include/drivers/coreboot/

F include/lib/coreboot.h

F lib/coreboot/

eMMC/UFS drivers

M Haojian Zhuang <haojian.zhuang@linaro.org>

G hzhuang1

F drivers/partition/

F drivers/synopsys/emmc/

F drivers/synopsys/ufs/

F drivers/ufs/

F include/drivers/dw_ufs.h

F include/drivers/ufs.h

F include/drivers/synopsys/dw_mmc.h

Power State Coordination Interface (PSCI)

M Javier Almansa Sobrino <Javier.AlmansaSobrino@arm.com>

G javieralso-arm

M Madhukar Pappireddy <Madhukar.Pappireddy@arm.com>

G madhukar-Arm

M Lauren Wehrmeister <Lauren.Wehrmeister@arm.com>

G laurenw-arm

M Zelalem Aweke <Zelalem.Aweke@arm.com>

G zelalem-aweke

F lib/psci/

DebugFS

M Olivier Deprez <olivier.deprez@arm.com>

G odeprez

F lib/debugfs/

Firmware Configuration Framework (FCONF)

M Madhukar Pappireddy <Madhukar.Pappireddy@arm.com>

G madhukar-Arm

M Manish Badarkhe <manish.badarkhe@arm.com>

G ManishVB-Arm

M Lauren Wehrmeister <Lauren.Wehrmeister@arm.com>

G laurenw-arm

F lib/fconf/

Performance Measurement Framework (PMF)

M Joao Alves <Joao.Alves@arm.com>

G J-Alves

M Jimmy Brisson <Jimmy.Brisson@arm.com>

G theotherjimmy

F lib/pmf/

Arm CPU libraries

M Lauren Wehrmeister <Lauren.Wehrmeister@arm.com>

G laurenw-arm

M John Powell <John.Powell@arm.com>

G john-powell-arm

F lib/cpus/

Reliability Availability Serviceabilty (RAS) framework

M Olivier Deprez <olivier.deprez@arm.com>

G odeprez

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

F lib/extensions/ras/

Activity Monitors Unit (AMU) extensions

M Alexei Fedorov <Alexei.Fedorov@arm.com>

G AlexeiFedorov

F lib/extensions/amu/

Memory Partitioning And Monitoring (MPAM) extensions

M Zelalem Aweke <Zelalem.Aweke@arm.com>

G zelalem-aweke

M Jimmy Brisson <Jimmy.Brisson@arm.com>

G theotherjimmy

F lib/extensions/mpam/

Pointer Authentication (PAuth) and Branch Target Identification (BTI) extensions

M Alexei Fedorov <Alexei.Fedorov@arm.com>

G AlexeiFedorov

M Zelalem Aweke <Zelalem.Aweke@arm.com>

G zelalem-aweke

F lib/extensions/pauth/

Statistical Profiling Extension (SPE)

M Zelalem Aweke <Zelalem.Aweke@arm.com>

G zelalem-aweke

M Jimmy Brisson <Jimmy.Brisson@arm.com>

G theotherjimmy

F lib/extensions/spe/

Scalable Vector Extension (SVE)

M Jimmy Brisson <Jimmy.Brisson@arm.com>

G theotherjimmy

F lib/extensions/sve/

Standard C library

M Alexei Fedorov <Alexei.Fedorov@arm.com>

G AlexeiFedorov

M John Powell <John.Powell@arm.com>

G john-powell-arm

F lib/libc/

Library At ROM (ROMlib)

M Madhukar Pappireddy <Madhukar.Pappireddy@arm.com>

G madhukar-Arm

F lib/romlib/

Translation tables (`xlat_tables`) library

M Javier Almansa Sobrino <Javier.AlmansaSobrino@arm.com>

G javieralso-arm

M Joao Alves <Joao.Alves@arm.com>

G J-Alves

F lib/xlat_tables_*/

IO abstraction layer

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

M Olivier Deprez <olivier.deprez@arm.com>

G odeprez

F drivers/io/

GIC driver

M Alexei Fedorov <Alexei.Fedorov@arm.com>

G AlexeiFedorov

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

M Madhukar Pappireddy <Madhukar.Pappireddy@arm.com>

G madhukar-Arm

M Olivier Deprez <olivier.deprez@arm.com>

G odeprez

F drivers/arm/gic/

Libfdt wrappers

M Madhukar Pappireddy <Madhukar.Pappireddy@arm.com>

G madhukar-Arm

M Manish Badarkhe <manish.badarkhe@arm.com>

G ManishVB-Arm

F common/fdt_wrappers.c

Firmware Encryption Framework

M Sumit Garg <sumit.garg@linaro.org>

G b49020

F drivers/io/io_encrypted.c

F include/drivers/io/io_encrypted.h

F include/tools_share/firmware_encrypted.h

Measured Boot

M Alexei Fedorov <Alexei.Fedorov@arm.com>

G AlexeiFedorov

M Javier Almansa Sobrino <Javier.AlmansaSobrino@arm.com>

G javieralso-arm

F drivers/measured_boot

F include/drivers/measured_boot

F plat/arm/board/fvp/fvp_measured_boot.c

Platform Ports

Allwinner ARMv8 platform port

M Andre Przywara <andre.przywara@arm.com>

G Andre-ARM

M Samuel Holland <samuel@sholland.org>

G smaeul

F docs/plat/allwinner.rst

F plat/allwinner/

F drivers/allwinner/

Amlogic Meson S905 (GXBB) platform port

M Andre Przywara <andre.przywara@arm.com>

G Andre-ARM

F docs/plat/meson-gxbb.rst

F drivers/amlogic/

F plat/amlogic/gxbb/

Amlogic Meson S905x (GXL) platform port

M Remi Pommarel <repk@triplefau.lt>

G remi-triplefault

F docs/plat/meson-gxl.rst

F plat/amlogic/gxl/

Amlogic Meson S905X2 (G12A) platform port

M Carlo Caione <ccaione@baylibre.com>

G carlocaione

F docs/plat/meson-g12a.rst

F plat/amlogic/g12a/

Amlogic Meson A113D (AXG) platform port

M Carlo Caione <ccaione@baylibre.com>

G carlocaione

F docs/plat/meson-axg.rst

F plat/amlogic/axg/

Arm FPGA platform port

M Andre Przywara <andre.przywara@arm.com>

G Andre-ARM

M Javier Almansa Sobrino <Javier.AlmansaSobrino@arm.com>

G javieralso-arm

F plat/arm/board/arm_fpga

Arm System Guidance for Infrastructure / Mobile FVP platforms

M Nariman Poushin <nariman.poushin@linaro.org>

G npoushin

M Thomas Abraham <thomas.abraham@arm.com>

G thomas-arm

F plat/arm/css/sgi/

F plat/arm/css/sgm/

F plat/arm/board/sgi575/

F plat/arm/board/sgm775/

HiSilicon HiKey and HiKey960 platform ports

M Haojian Zhuang <haojian.zhuang@linaro.org>

G hzhuang1

F docs/plat/hikey.rst

F docs/plat/hikey960.rst

F plat/hisilicon/hikey/

F plat/hisilicon/hikey960/

HiSilicon Poplar platform port

M Shawn Guo <shawn.guo@linaro.org>

G shawnguo2

F docs/plat/poplar.rst

F plat/hisilicon/poplar/

Intel SocFPGA platform ports

M Tien Hock Loh <tien.hock.loh@intel.com>

G thloh85-intel

M Hadi Asyrafi <muhammad.hadi.asyrafi.abdul.halim@intel.com>

G mabdulha

F plat/intel/soc

F drivers/intel/soc/

MediaTek platform ports

M Yidi Lin () <yidi.lin@mediatek.com>

G mtk09422

F plat/mediatek/

Marvell platform ports and SoC drivers

M Konstantin Porotchkin <kostap@marvell.com>

G kostapr

F docs/plat/marvell/

F plat/marvell/

F drivers/marvell/

F tools/marvell/

NVidia platform ports

M Varun Wadekar <vwadekar@nvidia.com>

G vwadekar

F docs/plat/nvidia-tegra.rst

F include/lib/cpus/aarch64/denver.h

F lib/cpus/aarch64/denver.S

F plat/nvidia/

NXP QorIQ Layerscape platform ports

M Jiafei Pan <jiafei.pan@nxp.com>

G qoriq-open-source

F docs/plat/ls1043a.rst

F plat/layerscape/

NXP i.MX 7 WaRP7 platform port and SoC drivers

M Bryan O'Donoghue <bryan.odonoghue@linaro.org>

G bryanodonoghue

M Jun Nie <jun.nie@linaro.org>

G niej

F docs/plat/warp7.rst

F plat/imx/common/

F plat/imx/imx7/

F drivers/imx/timer/

F drivers/imx/uart/

F drivers/imx/usdhc/

NXP i.MX 8 platform port

M Anson Huang <Anson.Huang@nxp.com>

G Anson-Huang

F docs/plat/imx8.rst

F plat/imx/

NXP i.MX8M platform port

M Jacky Bai <ping.bai@nxp.com>

G JackyBai

F docs/plat/imx8m.rst

F plat/imx/imx8m/

QEMU platform port

M Jens Wiklander <jens.wiklander@linaro.org>

G jenswi-linaro

F docs/plat/qemu.rst

F plat/qemu/

QTI platform port

M Saurabh Gorecha <sgorecha@codeaurora.org>

G sgorecha

M Debasish Mandal <dmandal@codeaurora.org>

M QTI TF Maintainers <qti.trustedfirmware.maintainers@codeaurora.org>

F docs/plat/qti.rst

F plat/qti/

Raspberry Pi 3 platform port

M Ying-Chun Liu (PaulLiu) <paul.liu@linaro.org>

G grandpaul

F docs/plat/rpi3.rst

F plat/rpi/rpi3/

F plat/rpi/common/

F drivers/rpi3/

F include/drivers/rpi3/

Raspberry Pi 4 platform port

M Andre Przywara <andre.przywara@arm.com>

G Andre-ARM

F docs/plat/rpi4.rst

F plat/rpi/rpi4/

F plat/rpi/common/

F drivers/rpi3/

F include/drivers/rpi3/

Renesas rcar-gen3 platform port

M Jorge Ramirez-Ortiz <jramirez@baylibre.com>

G ldts

M Marek Vasut <marek.vasut@gmail.com>

G marex

F docs/plat/rcar-gen3.rst

F plat/renesas/rcar

F drivers/renesas/rcar

F tools/renesas/rcar_layout_create

RockChip platform port

M Tony Xie <tony.xie@rock-chips.com>

G TonyXie06

G rockchip-linux

M Heiko Stuebner <heiko@sntech.de>

G mmind

F plat/rockchip/

STM32MP1 platform port

M Yann Gautier <yann.gautier@st.com>

G Yann-lms

F docs/plat/stm32mp1.rst

F drivers/st/

F fdts/stm32*

F include/drivers/st/

F include/dt-bindings/*/stm32*

F plat/st/

F tools/stm32image/

Synquacer platform port

M Sumit Garg <sumit.garg@linaro.org>

G b49020

F docs/plat/synquacer.rst

F plat/socionext/synquacer/

Texas Instruments platform port

M Andrew F. Davis <afd@ti.com>

G glneo

F docs/plat/ti-k3.rst

F plat/ti/

UniPhier platform port

M Orphan

F docs/plat/socionext-uniphier.rst

F plat/socionext/uniphier/

Xilinx platform port

M Siva Durga Prasad Paladugu <siva.durga.paladugu@xilinx.com>

G sivadur

F docs/plat/xilinx-zynqmp.rst

F plat/xilinx/

Secure Payloads and Dispatchers

OP-TEE dispatcher

M Jens Wiklander <jens.wiklander@linaro.org>

G jenswi-linaro

F docs/components/spd/optee-dispatcher.rst

F services/spd/opteed/

TLK/Trusty secure payloads

M Varun Wadekar <vwadekar@nvidia.com>

G vwadekar

F docs/components/spd/tlk-dispatcher.rst

F docs/components/spd/trusty-dispatcher.rst

F include/bl32/payloads/tlk.h

F services/spd/tlkd/

F services/spd/trusty/

Test Secure Payload (TSP)

M Manish Badarkhe <manish.badarkhe@arm.com>

G ManishVB-Arm

F bl32/tsp/

F services/spd/tspd/

Tools

Fiptool

M Joao Alves <Joao.Alves@arm.com>

G J-Alves

F tools/fiptool/

Cert_create tool

M Sandrine Bailleux <sandrine.bailleux@arm.com>

G sandrine-bailleux-arm

F tools/cert_create/

Encrypt_fw tool

M Sumit Garg <sumit.garg@linaro.org>

G b49020

F tools/encrypt_fw/

Sptool

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

F tools/sptool/

Build system

M Manish Pandey <manish.pandey2@arm.com>

G manish-pandey-arm

F Makefile

F make_helpers/

1.4 Support & Contact

We welcome any feedback on *TF-A* and there are several methods for providing it or for obtaining support.

Warning: If you think you have found a security vulnerability, please report this using the process defined in the *Security Handling* document.

1.4.1 Mailing Lists

Public mailing lists for TF-A and the wider Trusted Firmware project are hosted on TrustedFirmware.org. The mailing lists can be used for general enquiries, enhancement requests and issue reports, or to follow and participate in technical or organizational discussions around the project. These discussions include design proposals, advance notice of changes and upcoming events.

The relevant lists for the TF-A project are:

- TF-A development
- TF-A-Tests development

You can see a summary of all the lists on the TrustedFirmware.org website.

1.4.2 Open Tech Forum Call

Every other week, we organize a call with all interested TF-A contributors. Anyone is welcome to join. This is an opportunity to discuss any technical topic within the community. More details can be found here.

1.4.3 Issue Tracker

Bug reports may be filed on the issue tracker on the TrustedFirmware.org website. Using this tracker gives everyone visibility of the known issues in TF-A.

1.4.4 Arm Licensees

Arm licensees have an additional support conduit - they may contact Arm directly via their partner managers.

1.5 Contributor Acknowledgements

Note: This file is only relevant for legacy contributions, to acknowledge the specific contributors referred to in "Arm Limited and Contributors" copyright notices. As contributors are now encouraged to put their name or company name directly into the copyright notices, this file is not relevant for new contributions. See the *License* document for the correct template to use for new contributions.

- Linaro Limited
- Marvell International Ltd.
- NVIDIA Corporation
- NXP Semiconductors
- Socionext Inc.
- STMicroelectronics
- Xilinx, Inc.

CHAPTER

TWO

GETTING STARTED

2.1 Prerequisites

This document describes the software requirements for building *TF-A* for AArch32 and AArch64 target platforms.

It may possible to build *TF-A* with combinations of software packages that are different from those listed below, however only the software described in this document can be officially supported.

2.1.1 Build Host

TF-A can be built using either a Linux or a Windows machine as the build host.

A relatively recent Linux distribution is recommended for building *TF-A*. We have performed tests using Ubuntu 16.04 LTS (64-bit) but other distributions should also work fine as a base, provided that the necessary tools and libraries can be installed.

2.1.2 Toolchain

TF-A can be built with any of the following *cross-compiler* toolchains that target the Armv7-A or Armv8-A architectures:

- GCC >= 9.2-2019.12 (from the Arm Developer website)
- Clang >= 4.0
- Arm Compiler >= 6.0

In addition, a native compiler is required to build the supporting tools.

Note: The software has also been built on Windows 7 Enterprise SP1, using CMD.EXE, Cygwin, and Msys (MinGW) shells, using version 5.3.1 of the GNU toolchain.

Note: For instructions on how to select the cross compiler refer to *Performing an Initial Build*.

2.1.3 Software and Libraries

The following tools are required to obtain and build *TF-A*:

- An appropriate toolchain (see *Toolchain*)
- GNU Make
- Git

The following libraries must be available to build one or more components or supporting tools:

- OpenSSL >= 1.0.1

  Required to build the cert_create tool.

The following libraries are required for Trusted Board Boot support:

- mbed TLS == 2.24.0 (tag: `mbedtls-2.24.0`)

These tools are optional:

- Device Tree Compiler (DTC) >= 1.4.6

  Needed if you want to rebuild the provided Flattened Device Tree (FDT) source files (`.dts` files). DTC is available for Linux through the package repositories of most distributions.

- Arm Development Studio 5 (DS-5)

  The standard software package used for debugging software on Arm development platforms and *FVP* models.

Package Installation (Linux)

If you are using the recommended Ubuntu distribution then you can install the required packages with the following command:

```
sudo apt install build-essential git libssl-dev
```

The optional packages can be installed using:

```
sudo apt install device-tree-compiler
```

2.1.4 Supporting Files

TF-A has been tested with pre-built binaries and file systems from Linaro Release 19.06. Alternatively, you can build the binaries from source using instructions in *Performing an Initial Build*.

2.1.5 Getting the TF-A Source

Source code for *TF-A* is maintained in a Git repository hosted on TrustedFirmware.org. To clone this repository from the server, run the following in your shell:

```
git clone "https://review.trustedfirmware.org/TF-A/trusted-firmware-a" && (cd
↪"trusted-firmware-a" && mkdir -p .git/hooks && curl -Lo `git rev-parse --git-dir`/
↪hooks/commit-msg https://review.trustedfirmware.org/tools/hooks/commit-msg; chmod␣
↪+x `git rev-parse --git-dir`/hooks/commit-msg)
```

This will clone the Git repository also install a *commit hook* that automatically inserts appropriate *Change-Id:* lines at the end of your commit messages. These change IDs are required when committing changes that you intend to push for review via our Gerrit system.

You can read more about Git hooks in the *githooks* page of the Git documentation, available at: https://git-scm.com/docs/githooks

Alternatively, you can clone without the commit hook using:

```
git clone "https://review.trustedfirmware.org/TF-A/trusted-firmware-a"
```

2.2 Building Documentation

To create a rendered copy of this documentation locally you can use the Sphinx tool to build and package the plain-text documents into HTML-formatted pages.

If you are building the documentation for the first time then you will need to check that you have the required software packages, as described in the *Prerequisites* section that follows.

Note: An online copy of the documentation is available at https://www.trustedfirmware.org/docs/tf-a, if you want to view a rendered copy without doing a local build.

2.2.1 Prerequisites

For building a local copy of the *TF-A* documentation you will need, at minimum:

- Python 3 (3.5 or later)
- PlantUML (1.2017.15 or later)

Optionally, the Dia application can be installed if you need to edit existing `.dia` diagram files, or create new ones.

You must also install the Python modules that are specified in the `requirements.txt` file in the root of the `docs` directory. These modules can be installed using `pip3` (the Python Package Installer). Passing this requirements file as an argument to `pip3` automatically installs the specific module versions required by *TF-A*.

An example set of installation commands for Ubuntu 18.04 LTS follows, assuming that the working directory is `docs`:

```
sudo apt install python3 python3-pip plantuml [dia]
pip3 install [--user] -r requirements.txt
```

Note: Several other modules will be installed as dependencies. Please review the list to ensure that there will be no conflicts with other modules already installed in your environment.

Passing the optional `--user` argument to `pip3` will install the Python packages only for the current user. Omitting this argument will attempt to install the packages globally and this will likely require the command to be run as root or using `sudo`.

Note: More advanced usage instructions for *pip* are beyond the scope of this document but you can refer to the pip homepage for detailed guides.

2.2.2 Building rendered documentation

Documents can be built into HTML-formatted pages from project root directory by running the following command.

```
make doc
```

Output from the build process will be placed in:

```
docs/build/html
```

We also support building documentation in other formats. From the `docs` directory of the project, run the following command to see the supported formats. It is important to note that you will not get the correct result if the command is run from the project root directory, as that would invoke the top-level Makefile for *TF-A* itself.

```
make help
```

2.2.3 Building rendered documentation from a container

There may be cases where you can not either install or upgrade required dependencies to generate the documents, so in this case, one way to create the documentation is through a docker container. The first step is to check if docker is installed in your host, otherwise check main docker page for installation instructions. Once installed, run the following script from project root directory

```
docker run --rm -v $PWD:/TF sphinxdoc/sphinx \
       bash -c 'cd /TF && \
       pip3 install plantuml -r ./docs/requirements.txt && make doc'
```

The above command fetches the `sphinxdoc/sphinx` container from docker hub, launches the container, installs documentation requirements and finally creates the documentation. Once done, exit the container and output from the build process will be placed in:

```
docs/build/html
```

2.3 Building Supporting Tools

2.3.1 Building and using the FIP tool

Firmware Image Package (FIP) is a packaging format used by TF-A to package firmware images in a single binary. The number and type of images that should be packed in a FIP is platform specific and may include TF-A images and other firmware images required by the platform. For example, most platforms require a BL33 image which corresponds to the normal world bootloader (e.g. UEFI or U-Boot).

The TF-A build system provides the make target `fip` to create a FIP file for the specified platform using the FIP creation tool included in the TF-A project. Examples below show how to build a FIP file for FVP, packaging TF-A and BL33 images.

For AArch64:

```
make PLAT=fvp BL33=<path-to>/bl33.bin fip
```

For AArch32:

```
make PLAT=fvp ARCH=aarch32 AARCH32_SP=sp_min BL33=<path-to>/bl33.bin fip
```

The resulting FIP may be found in:

```
build/fvp/<build-type>/fip.bin
```

For advanced operations on FIP files, it is also possible to independently build the tool and create or modify FIPs using this tool. To do this, follow these steps:

It is recommended to remove old artifacts before building the tool:

```
make -C tools/fiptool clean
```

Build the tool:

```
make [DEBUG=1] [V=1] fiptool
```

The tool binary can be located in:

```
./tools/fiptool/fiptool
```

Invoking the tool with `help` will print a help message with all available options.

Example 1: create a new Firmware package `fip.bin` that contains BL2 and BL31:

```
./tools/fiptool/fiptool create \
    --tb-fw build/<platform>/<build-type>/bl2.bin \
    --soc-fw build/<platform>/<build-type>/bl31.bin \
    fip.bin
```

Example 2: view the contents of an existing Firmware package:

```
./tools/fiptool/fiptool info <path-to>/fip.bin
```

Example 3: update the entries of an existing Firmware package:

```
# Change the BL2 from Debug to Release version
./tools/fiptool/fiptool update \
    --tb-fw build/<platform>/release/bl2.bin \
    build/<platform>/debug/fip.bin
```

Example 4: unpack all entries from an existing Firmware package:

```
# Images will be unpacked to the working directory
./tools/fiptool/fiptool unpack <path-to>/fip.bin
```

Example 5: remove an entry from an existing Firmware package:

```
./tools/fiptool/fiptool remove \
    --tb-fw build/<platform>/debug/fip.bin
```

Note that if the destination FIP file exists, the create, update and remove operations will automatically overwrite it.

The unpack operation will fail if the images already exist at the destination. In that case, use -f or –force to continue.

More information about FIP can be found in the *Firmware Design* document.

2.3.2 Building the Certificate Generation Tool

The `cert_create` tool is built as part of the TF-A build process when the `fip` make target is specified and TBB is enabled (as described in the previous section), but it can also be built separately with the following command:

```
make PLAT=<platform> [DEBUG=1] [V=1] certtool
```

For platforms that require their own IDs in certificate files, the generic 'cert_create' tool can be built with the following command. Note that the target platform must define its IDs within a `platform_oid.h` header file for the build to succeed.

```
make PLAT=<platform> USE_TBBR_DEFS=0 [DEBUG=1] [V=1] certtool
```

`DEBUG=1` builds the tool in debug mode. `V=1` makes the build process more verbose. The following command should be used to obtain help about the tool:

```
./tools/cert_create/cert_create -h
```

Building the Firmware Encryption Tool

The `encrypt_fw` tool is built as part of the TF-A build process when the `fip` make target is specified, DECRYPTION_SUPPORT and TBB are enabled, but it can also be built separately with the following command:

```
make PLAT=<platform> [DEBUG=1] [V=1] enctool
```

`DEBUG=1` builds the tool in debug mode. `V=1` makes the build process more verbose. The following command should be used to obtain help about the tool:

```
./tools/encrypt_fw/encrypt_fw -h
```

Note that the enctool in its current implementation only supports encryption key to be provided in plain format. A typical implementation can very well extend this tool to support custom techniques to protect encryption key.

Also, a user may choose to provide encryption key or nonce as an input file via using `cat <filename>` instead of a hex string.

2.4 Performing an Initial Build

- Before building TF-A, the environment variable `CROSS_COMPILE` must point to the Linaro cross compiler.

  For AArch64:

  ```
  export CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-none-elf-
  ```

  For AArch32:

  ```
  export CROSS_COMPILE=<path-to-aarch32-gcc>/bin/arm-none-eabi-
  ```

  It is possible to build TF-A using Clang or Arm Compiler 6. To do so `CC` needs to point to the clang or armclang binary, which will also select the clang or armclang assembler. Be aware that for Arm Compiler, the GNU linker is used by default. However for Clang LLVM linker (LLD) is used by default. In case of being needed the linker can be overridden using the `LD` variable. LLVM linker (LLD) version 9 is known to work with TF-A.

  In both cases `CROSS_COMPILE` should be set as described above.

  Arm Compiler 6 will be selected when the base name of the path assigned to `CC` matches the string 'armclang'.

  For AArch64 using Arm Compiler 6:

  ```
  export CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-none-elf-
  make CC=<path-to-armclang>/bin/armclang PLAT=<platform> all
  ```

  Clang will be selected when the base name of the path assigned to `CC` contains the string 'clang'. This is to allow both clang and clang-X.Y to work.

  For AArch64 using clang:

  ```
  export CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-none-elf-
  make CC=<path-to-clang>/bin/clang PLAT=<platform> all
  ```

- Change to the root directory of the TF-A source tree and build.

  For AArch64:

  ```
  make PLAT=<platform> all
  ```

  For AArch32:

  ```
  make PLAT=<platform> ARCH=aarch32 AARCH32_SP=sp_min all
  ```

  Notes:

  – If `PLAT` is not specified, `fvp` is assumed by default. See the *Build Options* document for more information on available build options.

  – (AArch32 only) Currently only `PLAT=fvp` is supported.

  – (AArch32 only) `AARCH32_SP` is the AArch32 EL3 Runtime Software and it corresponds to the BL32 image. A minimal `AARCH32_SP`, sp_min, is provided by TF-A to demonstrate how PSCI Library can be integrated with an AArch32 EL3 Runtime Software. Some AArch32 EL3 Runtime Software may include other runtime services, for example Trusted OS services. A guide to integrate PSCI library with AArch32 EL3 Runtime Software can be found at *PSCI Library Integration guide for Armv8-A AArch32 systems*.

  – (AArch64 only) The TSP (Test Secure Payload), corresponding to the BL32 image, is not compiled in by default. Refer to the *Test Secure Payload (TSP) and Dispatcher (TSPD)* document for details on building the TSP.

  - By default this produces a release version of the build. To produce a debug version instead, refer to the "Debugging options" section below.
  - The build process creates products in a `build` directory tree, building the objects and binaries for each boot loader stage in separate sub-directories. The following boot loader binary files are created from the corresponding ELF files:
    - `build/<platform>/<build-type>/bl1.bin`
    - `build/<platform>/<build-type>/bl2.bin`
    - `build/<platform>/<build-type>/bl31.bin` (AArch64 only)
    - `build/<platform>/<build-type>/bl32.bin` (mandatory for AArch32)

    where `<platform>` is the name of the chosen platform and `<build-type>` is either `debug` or `release`. The actual number of images might differ depending on the platform.

- Build products for a specific build variant can be removed using:

```
make DEBUG=<D> PLAT=<platform> clean
```

  ... where `<D>` is `0` or `1`, as specified when building.

  The build tree can be removed completely using:

```
make realclean
```

2.5 Build Options

The TF-A build system supports the following build options. Unless mentioned otherwise, these options are expected to be specified at the build command line and are not to be modified in any component makefiles. Note that the build system doesn't track dependency for build options. Therefore, if any of the build options are changed from a previous build, a clean build must be performed.

2.5.1 Common build options

- `AARCH32_INSTRUCTION_SET`: Choose the AArch32 instruction set that the compiler should use. Valid values are T32 and A32. It defaults to T32 due to code having a smaller resulting size.
- `AARCH32_SP` : Choose the AArch32 Secure Payload component to be built as as the BL32 image when `ARCH=aarch32`. The value should be the path to the directory containing the SP source, relative to the `bl32/`; the directory is expected to contain a makefile called `<aarch32_sp-value>.mk`.
- `ARCH` : Choose the target build architecture for TF-A. It can take either `aarch64` or `aarch32` as values. By default, it is defined to `aarch64`.
- `ARM_ARCH_MAJOR`: The major version of Arm Architecture to target when compiling TF-A. Its value must be numeric, and defaults to 8 . See also, *Armv8 Architecture Extensions* and *Armv7 Architecture Extensions* in *Firmware Design*.
- `ARM_ARCH_MINOR`: The minor version of Arm Architecture to target when compiling TF-A. Its value must be a numeric, and defaults to 0. See also, *Armv8 Architecture Extensions* in *Firmware Design*.

- `BL2`: This is an optional build option which specifies the path to BL2 image for the `fip` target. In this case, the BL2 in the TF-A will not be built.
- `BL2U`: This is an optional build option which specifies the path to BL2U image. In this case, the BL2U in TF-A will not be built.
- `BL2_AT_EL3`: This is an optional build option that enables the use of BL2 at EL3 execution level.
- `BL2_IN_XIP_MEM`: In some use-cases BL2 will be stored in eXecute In Place (XIP) memory, like BL1. In these use-cases, it is necessary to initialize the RW sections in RAM, while leaving the RO sections in place. This option enable this use-case. For now, this option is only supported when BL2_AT_EL3 is set to '1'.
- `BL31`: This is an optional build option which specifies the path to BL31 image for the `fip` target. In this case, the BL31 in TF-A will not be built.
- `BL31_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the BL31 private key in PEM format. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `BL32`: This is an optional build option which specifies the path to BL32 image for the `fip` target. In this case, the BL32 in TF-A will not be built.
- `BL32_EXTRA1`: This is an optional build option which specifies the path to Trusted OS Extra1 image for the `fip` target.
- `BL32_EXTRA2`: This is an optional build option which specifies the path to Trusted OS Extra2 image for the `fip` target.
- `BL32_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the BL32 private key in PEM format. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `BL33`: Path to BL33 image in the host file system. This is mandatory for `fip` target in case TF-A BL2 is used.
- `BL33_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the BL33 private key in PEM format. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `BRANCH_PROTECTION`: Numeric value to enable ARMv8.3 Pointer Authentication and ARMv8.5 Branch Target Identification support for TF-A BL images themselves. If enabled, it is needed to use a compiler that supports the option `-mbranch-protection`. Selects the branch protection features to use:
- 0: Default value turns off all types of branch protection
- 1: Enables all types of branch protection features
- 2: Return address signing to its standard level
- 3: Extend the signing to include leaf functions
- 4: Turn on branch target identification mechanism

  The table below summarizes `BRANCH_PROTECTION` values, GCC compilation options and resulting PAuth/BTI features.

Value	GCC option	PAuth	BTI
0	none	N	N
1	standard	Y	Y
2	pac-ret	Y	N
3	pac-ret+leaf	Y	N
4	bti	N	Y

This option defaults to 0 and this is an experimental feature. Note that Pointer Authentication is enabled for Non-secure world irrespective of the value of this option if the CPU supports it.

- BUILD_MESSAGE_TIMESTAMP: String used to identify the time and date of the compilation of each build. It must be set to a C string (including quotes where applicable). Defaults to a string that contains the time and date of the compilation.
- BUILD_STRING: Input string for VERSION_STRING, which allows the TF-A build to be uniquely identified. Defaults to the current git commit id.
- BUILD_BASE: Output directory for the build. Defaults to ./build
- CFLAGS: Extra user options appended on the compiler's command line in addition to the options set by the build system.
- COLD_BOOT_SINGLE_CPU: This option indicates whether the platform may release several CPUs out of reset. It can take either 0 (several CPUs may be brought up) or 1 (only one CPU will ever be brought up during cold reset). Default is 0. If the platform always brings up a single CPU, there is no need to distinguish between primary and secondary CPUs and the boot path can be optimised. The plat_is_my_cpu_primary() and plat_secondary_cold_boot_setup() platform porting interfaces do not need to be implemented in this case.
- COT: When Trusted Boot is enabled, selects the desired chain of trust. Defaults to tbbr.
- CRASH_REPORTING: A non-zero value enables a console dump of processor register state when an unexpected exception occurs during execution of BL31. This option defaults to the value of DEBUG - i.e. by default this is only enabled for a debug build of the firmware.
- CREATE_KEYS: This option is used when GENERATE_COT=1. It tells the certificate generation tool to create new keys in case no valid keys are present or specified. Allowed options are '0' or '1'. Default is '1'.
- CTX_INCLUDE_AARCH32_REGS : Boolean option that, when set to 1, will cause the AArch32 system registers to be included when saving and restoring the CPU context. The option must be set to 0 for AArch64-only platforms (that is on hardware that does not implement AArch32, or at least not at EL1 and higher ELs). Default value is 1.
- CTX_INCLUDE_EL2_REGS : This boolean option provides context save/restore operations when entering/exiting an EL2 execution context. This is of primary interest when Armv8.4-SecEL2 extension is implemented. Default is 0 (disabled). This option must be equal to 1 (enabled) when SPD=spmd and SPMD_SPM_AT_SEL2 is set.
- CTX_INCLUDE_FPREGS: Boolean option that, when set to 1, will cause the FP registers to be included when saving and restoring the CPU context. Default is 0.
- CTX_INCLUDE_NEVE_REGS: Boolean option that, when set to 1, will cause the Armv8.4-NV registers to be saved/restored when entering/exiting an EL2 execution context. Default value is 0.
- CTX_INCLUDE_PAUTH_REGS: Boolean option that, when set to 1, enables Pointer Authentication for Secure world. This will cause the ARMv8.3-PAuth registers to be included when saving and restoring the CPU context as part of world switch. Default value is 0 and this is an experimental feature. Note that Pointer Authentication is enabled for Non-secure world irrespective of the value of this flag if the CPU supports it.
- DEBUG: Chooses between a debug and release build. It can take either 0 (release) or 1 (debug) as values. 0 is the default.
- DECRYPTION_SUPPORT: This build flag enables the user to select the authenticated decryption algorithm to be used to decrypt firmware/s during boot. It accepts 2 values: aes_gcm and none. The default value of this flag is none to disable firmware decryption which is an optional feature as per TBBR. Also, it is an experimental feature.
- DISABLE_BIN_GENERATION: Boolean option to disable the generation of the binary image. If set to 1, then only the ELF image is built. 0 is the default.

- `DYN_DISABLE_AUTH`: Provides the capability to dynamically disable Trusted Board Boot authentication at runtime. This option is meant to be enabled only for development platforms. `TRUSTED_BOARD_BOOT` flag must be set if this flag has to be enabled. 0 is the default.
- `E`: Boolean option to make warnings into errors. Default is 1.
- `EL3_PAYLOAD_BASE`: This option enables booting an EL3 payload instead of the normal boot flow. It must specify the entry point address of the EL3 payload. Please refer to the "Booting an EL3 payload" section for more details.
- `ENABLE_AMU`: Boolean option to enable Activity Monitor Unit extensions. This is an optional architectural feature available on v8.4 onwards. Some v8.2 implementations also implement an AMU and this option can be used to enable this feature on those systems as well. Default is 0.
- `ENABLE_ASSERTIONS`: This option controls whether or not calls to `assert()` are compiled out. For debug builds, this option defaults to 1, and calls to `assert()` are left in place. For release builds, this option defaults to 0 and calls to `assert()` function are compiled out. This option can be set independently of `DEBUG`. It can also be used to hide any auxiliary code that is only required for the assertion and does not fit in the assertion itself.
- `ENABLE_BACKTRACE`: This option controls whether to enable backtrace dumps or not. It is supported in both AArch64 and AArch32. However, in AArch32 the format of the frame records are not defined in the AAPCS and they are defined by the implementation. This implementation of backtrace only supports the format used by GCC when T32 interworking is disabled. For this reason enabling this option in AArch32 will force the compiler to only generate A32 code. This option is enabled by default only in AArch64 debug builds, but this behaviour can be overridden in each platform's Makefile or in the build command line.
- `ENABLE_LTO`: Boolean option to enable Link Time Optimization (LTO) support in GCC for TF-A. This option is currently only supported for AArch64. Default is 0.
- `ENABLE_MPAM_FOR_LOWER_ELS`: Boolean option to enable lower ELs to use MPAM feature. MPAM is an optional Armv8.4 extension that enables various memory system components and resources to define partitions; software running at various ELs can assign themselves to desired partition to control their performance aspects.

  When this option is set to `1`, EL3 allows lower ELs to access their own MPAM registers without trapping into EL3. This option doesn't make use of partitioning in EL3, however. Platform initialisation code should configure and use partitions in EL3 as required. This option defaults to `0`.
- `ENABLE_PIE`: Boolean option to enable Position Independent Executable(PIE) support within generic code in TF-A. This option is currently only supported in BL2_AT_EL3, BL31, and BL32 (TSP). Default is 0.
- `ENABLE_PMF`: Boolean option to enable support for optional Performance Measurement Framework(PMF). Default is 0.
- `ENABLE_PSCI_STAT`: Boolean option to enable support for optional PSCI functions `PSCI_STAT_RESIDENCY` and `PSCI_STAT_COUNT`. Default is 0. In the absence of an alternate stat collection backend, `ENABLE_PMF` must be enabled. If `ENABLE_PMF` is set, the residency statistics are tracked in software.
- `ENABLE_RUNTIME_INSTRUMENTATION`: Boolean option to enable runtime instrumentation which injects timestamp collection points into TF-A to allow runtime performance to be measured. Currently, only PSCI is instrumented. Enabling this option enables the `ENABLE_PMF` build option as well. Default is 0.
- `ENABLE_SPE_FOR_LOWER_ELS` : Boolean option to enable Statistical Profiling extensions. This is an optional architectural feature for AArch64. The default is 1 but is automatically disabled when the target architecture is AArch32.
- `ENABLE_SVE_FOR_NS`: Boolean option to enable Scalable Vector Extension (SVE) for the Non-secure world only. SVE is an optional architectural feature for AArch64. Note that when SVE is enabled for the Non-secure world, access to SIMD and floating-point functionality from the Secure world is disabled. This is to avoid corruption of the Non-secure world data in the Z-registers which are aliased by the SIMD and FP registers.

The build option is not compatible with the CTX_INCLUDE_FPREGS build option, and will raise an assert on platforms where SVE is implemented and ENABLE_SVE_FOR_NS set to 1. The default is 1 but is automatically disabled when the target architecture is AArch32.

- ENABLE_STACK_PROTECTOR: String option to enable the stack protection checks in GCC. Allowed values are "all", "strong", "default" and "none". The default value is set to "none". "strong" is the recommended stack protection level if this feature is desired. "none" disables the stack protection. For all values other than "none", the plat_get_stack_protector_canary() platform hook needs to be implemented. The value is passed as the last component of the option -fstack-protector-$ENABLE_STACK_PROTECTOR.
- ENCRYPT_BL31: Binary flag to enable encryption of BL31 firmware. This flag depends on DECRYPTION_SUPPORT build flag which is marked as experimental.
- ENCRYPT_BL32: Binary flag to enable encryption of Secure BL32 payload. This flag depends on DECRYPTION_SUPPORT build flag which is marked as experimental.
- ENC_KEY: A 32-byte (256-bit) symmetric key in hex string format. It could either be SSK or BSSK depending on FW_ENC_STATUS flag. This value depends on DECRYPTION_SUPPORT build flag which is marked as experimental.
- ENC_NONCE: A 12-byte (96-bit) encryption nonce or Initialization Vector (IV) in hex string format. This value depends on DECRYPTION_SUPPORT build flag which is marked as experimental.
- ERROR_DEPRECATED: This option decides whether to treat the usage of deprecated platform APIs, helper functions or drivers within Trusted Firmware as error. It can take the value 1 (flag the use of deprecated APIs as error) or 0. The default is 0.
- EL3_EXCEPTION_HANDLING: When set to 1, enable handling of exceptions targeted at EL3. When set 0 (default), no exceptions are expected or handled at EL3, and a panic will result. This is supported only for AArch64 builds.
- EVENT_LOG_LEVEL: Chooses the log level to use for Measured Boot when MEASURED_BOOT is enabled. For a list of valid values, see LOG_LEVEL. Default value is 40 (LOG_LEVEL_INFO).
- FAULT_INJECTION_SUPPORT: ARMv8.4 extensions introduced support for fault injection from lower ELs, and this build option enables lower ELs to use Error Records accessed via System Registers to inject faults. This is applicable only to AArch64 builds.

  This feature is intended for testing purposes only, and is advisable to keep disabled for production images.
- FIP_NAME: This is an optional build option which specifies the FIP filename for the fip target. Default is fip.bin.
- FWU_FIP_NAME: This is an optional build option which specifies the FWU FIP filename for the fwu_fip target. Default is fwu_fip.bin.
- FW_ENC_STATUS: Top level firmware's encryption numeric flag, values:

```
0: Encryption is done with Secret Symmetric Key (SSK) which is common
   for a class of devices.
1: Encryption is done with Binding Secret Symmetric Key (BSSK) which is
   unique per device.
```

  This flag depends on DECRYPTION_SUPPORT build flag which is marked as experimental.
- GENERATE_COT: Boolean flag used to build and execute the cert_create tool to create certificates as per the Chain of Trust described in *Trusted Board Boot*. The build system then calls fiptool to include the certificates in the FIP and FWU_FIP. Default value is '0'.

  Specify both TRUSTED_BOARD_BOOT=1 and GENERATE_COT=1 to include support for the Trusted Board Boot feature in the BL1 and BL2 images, to generate the corresponding certificates, and to include those certificates in the FIP and FWU_FIP.

Note that if TRUSTED_BOARD_BOOT=0 and GENERATE_COT=1, the BL1 and BL2 images will not include support for Trusted Board Boot. The FIP will still include the corresponding certificates. This FIP can be used to verify the Chain of Trust on the host machine through other mechanisms.

Note that if TRUSTED_BOARD_BOOT=1 and GENERATE_COT=0, the BL1 and BL2 images will include support for Trusted Board Boot, but the FIP and FWU_FIP will not include the corresponding certificates, causing a boot failure.

- GICV2_G0_FOR_EL3: Unlike GICv3, the GICv2 architecture doesn't have inherent support for specific EL3 type interrupts. Setting this build option to 1 assumes GICv2 *Group 0* interrupts are expected to target EL3, both by *platform abstraction layer* and *Interrupt Management Framework*. This allows GICv2 platforms to enable features requiring EL3 interrupt type. This also means that all GICv2 Group 0 interrupts are delivered to EL3, and the Secure Payload interrupts needs to be synchronously handed over to Secure EL1 for handling. The default value of this option is 0, which means the Group 0 interrupts are assumed to be handled by Secure EL1.
- HANDLE_EA_EL3_FIRST: When set to 1, External Aborts and SError Interrupts will be always trapped in EL3 i.e. in BL31 at runtime. When set to 0 (default), these exceptions will be trapped in the current exception level (or in EL1 if the current exception level is EL0).
- HW_ASSISTED_COHERENCY: On most Arm systems to-date, platform-specific software operations are required for CPUs to enter and exit coherency. However, newer systems exist where CPUs' entry to and exit from coherency is managed in hardware. Such systems require software to only initiate these operations, and the rest is managed in hardware, minimizing active software management. In such systems, this boolean option enables TF-A to carry out build and run-time optimizations during boot and power management operations. This option defaults to 0 and if it is enabled, then it implies WARMBOOT_ENABLE_DCACHE_EARLY is also enabled.

  If this flag is disabled while the platform which TF-A is compiled for includes cores that manage coherency in hardware, then a compilation error is generated. This is based on the fact that a system cannot have, at the same time, cores that manage coherency in hardware and cores that don't. In other words, a platform cannot have, at the same time, cores that require HW_ASSISTED_COHERENCY=1 and cores that require HW_ASSISTED_COHERENCY=0.

  Note that, when HW_ASSISTED_COHERENCY is enabled, version 2 of translation library (xlat tables v2) must be used; version 1 of translation library is not supported.
- INVERTED_MEMMAP: memmap tool print by default lower addresses at the bottom, higher addresses at the top. This buid flag can be set to '1' to invert this behavior. Lower addresses will be printed at the top and higher addresses at the bottom.
- JUNO_AARCH32_EL3_RUNTIME: This build flag enables you to execute EL3 runtime software in AArch32 mode, which is required to run AArch32 on Juno. By default this flag is set to '0'. Enabling this flag builds BL1 and BL2 in AArch64 and facilitates the loading of SP_MIN and BL33 as AArch32 executable images.
- KEY_ALG: This build flag enables the user to select the algorithm to be used for generating the PKCS keys and subsequent signing of the certificate. It accepts 3 values: rsa, rsa_1_5 and ecdsa. The option rsa_1_5 is the legacy PKCS#1 RSA 1.5 algorithm which is not TBBR compliant and is retained only for compatibility. The default value of this flag is rsa which is the TBBR compliant PKCS#1 RSA 2.1 scheme.
- KEY_SIZE: This build flag enables the user to select the key size for the algorithm specified by KEY_ALG. The valid values for KEY_SIZE depend on the chosen algorithm and the cryptographic module.

KEY_ALG	Possible key sizes
rsa	1024 , 2048 (default), 3072, 4096*
ecdsa	unavailable

  - Only 2048 bits size is available with CryptoCell 712 SBROM release 1. Only 3072 bits size is available with CryptoCell 712 SBROM release 2.

- `HASH_ALG`: This build flag enables the user to select the secure hash algorithm. It accepts 3 values: `sha256`, `sha384` and `sha512`. The default value of this flag is `sha256`.
- `LDFLAGS`: Extra user options appended to the linkers' command line in addition to the one set by the build system.
- `LOG_LEVEL`: Chooses the log level, which controls the amount of console log output compiled into the build. This should be one of the following:

  ```
  0  (LOG_LEVEL_NONE)
  10 (LOG_LEVEL_ERROR)
  20 (LOG_LEVEL_NOTICE)
  30 (LOG_LEVEL_WARNING)
  40 (LOG_LEVEL_INFO)
  50 (LOG_LEVEL_VERBOSE)
  ```

  All log output up to and including the selected log level is compiled into the build. The default value is 40 in debug builds and 20 in release builds.
- `MEASURED_BOOT`: Boolean flag to include support for the Measured Boot feature. If this flag is enabled `TRUSTED_BOARD_BOOT` must be set. This option defaults to 0 and is an experimental feature in the stage of development.
- `NON_TRUSTED_WORLD_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the Non-Trusted World private key in PEM format. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `NS_BL2U`: Path to NS_BL2U image in the host file system. This image is optional. It is only needed if the platform makefile specifies that it is required in order to build the `fwu_fip` target.
- `NS_TIMER_SWITCH`: Enable save and restore for non-secure timer register contents upon world switch. It can take either 0 (don't save and restore) or 1 (do save and restore). 0 is the default. An SPD may set this to 1 if it wants the timer registers to be saved and restored.
- `OVERRIDE_LIBC`: This option allows platforms to override the default libc for the BL image. It can be either 0 (include) or 1 (remove). The default value is 0.
- `PL011_GENERIC_UART`: Boolean option to indicate the PL011 driver that the underlying hardware is not a full PL011 UART but a minimally compliant generic UART, which is a subset of the PL011. The driver will not access any register that is not part of the SBSA generic UART specification. Default value is 0 (a full PL011 compliant UART is present).
- `PLAT`: Choose a platform to build TF-A for. The chosen platform name must be subdirectory of any depth under `plat/`, and must contain a platform makefile named `platform.mk`. For example, to build TF-A for the Arm Juno board, select PLAT=juno.
- `PRELOADED_BL33_BASE`: This option enables booting a preloaded BL33 image instead of the normal boot flow. When defined, it must specify the entry point address for the preloaded BL33 image. This option is incompatible with `EL3_PAYLOAD_BASE`. If both are defined, `EL3_PAYLOAD_BASE` has priority over `PRELOADED_BL33_BASE`.
- `PROGRAMMABLE_RESET_ADDRESS`: This option indicates whether the reset vector address can be programmed or is fixed on the platform. It can take either 0 (fixed) or 1 (programmable). Default is 0. If the platform has a programmable reset address, it is expected that a CPU will start executing code directly at the right address, both on a cold and warm reset. In this case, there is no need to identify the entrypoint on boot and the boot path can be optimised. The `plat_get_my_entrypoint()` platform porting interface does not need to be implemented in this case.
- `PSCI_EXTENDED_STATE_ID`: As per PSCI1.0 Specification, there are 2 formats possible for the PSCI power-state parameter: original and extended State-ID formats. This flag if set to 1, configures the generic PSCI layer to use the extended format. The default value of this flag is 0, which means by default the original

power-state format is used by the PSCI implementation. This flag should be specified by the platform makefile and it governs the return value of PSCI_FEATURES API for CPU_SUSPEND smc function id. When this option is enabled on Arm platforms, the option `ARM_RECOM_STATE_ID_ENC` needs to be set to 1 as well.

- `RAS_EXTENSION`: When set to `1`, enable Armv8.2 RAS features. RAS features are an optional extension for pre-Armv8.2 CPUs, but are mandatory for Armv8.2 or later CPUs.

  When `RAS_EXTENSION` is set to `1`, `HANDLE_EA_EL3_FIRST` must also be set to `1`.

  This option is disabled by default.

- `RESET_TO_BL31`: Enable BL31 entrypoint as the CPU reset vector instead of the BL1 entrypoint. It can take the value 0 (CPU reset to BL1 entrypoint) or 1 (CPU reset to BL31 entrypoint). The default value is 0.
- `RESET_TO_SP_MIN`: SP_MIN is the minimal AArch32 Secure Payload provided in TF-A. This flag configures SP_MIN entrypoint as the CPU reset vector instead of the BL1 entrypoint. It can take the value 0 (CPU reset to BL1 entrypoint) or 1 (CPU reset to SP_MIN entrypoint). The default value is 0.
- `ROT_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the ROT private key in PEM format and enforces public key hash generation. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `SAVE_KEYS`: This option is used when `GENERATE_COT=1`. It tells the certificate generation tool to save the keys used to establish the Chain of Trust. Allowed options are '0' or '1'. Default is '0' (do not save).
- `SCP_BL2`: Path to SCP_BL2 image in the host file system. This image is optional. If a SCP_BL2 image is present then this option must be passed for the `fip` target.
- `SCP_BL2_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the SCP_BL2 private key in PEM format. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `SCP_BL2U`: Path to SCP_BL2U image in the host file system. This image is optional. It is only needed if the platform makefile specifies that it is required in order to build the `fwu_fip` target.
- `SDEI_SUPPORT`: Setting this to `1` enables support for Software Delegated Exception Interface to BL31 image. This defaults to `0`.

  When set to `1`, the build option `EL3_EXCEPTION_HANDLING` must also be set to `1`.

- `SEPARATE_CODE_AND_RODATA`: Whether code and read-only data should be isolated on separate memory pages. This is a trade-off between security and memory usage. See "Isolating code and read-only data on separate memory pages" section in *Firmware Design*. This flag is disabled by default and affects all BL images.
- `SEPARATE_NOBITS_REGION`: Setting this option to `1` allows the NOBITS sections of BL31 (.bss, stacks, page tables, and coherent memory) to be allocated in RAM discontiguous from the loaded firmware image. When set, the platform is expected to provide definitons for `BL31_NOBITS_BASE` and `BL31_NOBITS_LIMIT`. When the option is `0` (the default), NOBITS sections are placed in RAM immediately following the loaded firmware image.
- `SPD`: Choose a Secure Payload Dispatcher component to be built into TF-A. This build option is only valid if `ARCH=aarch64`. The value should be the path to the directory containing the SPD source, relative to `services/spd/`; the directory is expected to contain a makefile called `<spd-value>.mk`. The SPM Dispatcher standard service is located in services/std_svc/spmd and enabled by `SPD=spmd`. The SPM Dispatcher cannot be enabled when the `SPM_MM` option is enabled.
- `SPIN_ON_BL1_EXIT`: This option introduces an infinite loop in BL1. It can take either 0 (no loop) or 1 (add a loop). 0 is the default. This loop stops execution in BL1 just before handing over to BL31. At this point, all firmware images have been loaded in memory, and the MMU and caches are turned off. Refer to the "Debugging options" section for more details.
- `SPMD_SPM_AT_SEL2` : this boolean option is used jointly with the SPM Dispatcher option (`SPD=spmd`). When enabled (1) it indicates the SPMC component runs at the S-EL2 execution state provided by the Armv8.4-

SecEL2 extension. This is the default when enabling the SPM Dispatcher. When disabled (0) it indicates the SPMC component runs at the S-EL1 execution state. This latter configuration supports pre-Armv8.4 platforms (aka not implementing the Armv8.4-SecEL2 extension).

- `SPM_MM` : Boolean option to enable the Management Mode (MM)-based Secure Partition Manager (SPM) implementation. The default value is `0` (disabled). This option cannot be enabled (`1`) when SPM Dispatcher is enabled (`SPD=spmd`).
- `SP_LAYOUT_FILE`: Platform provided path to JSON file containing the description of secure partitions. The build system will parse this file and package all secure partition blobs into the FIP. This file is not necessarily part of TF-A tree. Only available when `SPD=spmd`.
- `SP_MIN_WITH_SECURE_FIQ`: Boolean flag to indicate the SP_MIN handles secure interrupts (caught through the FIQ line). Platforms can enable this directive if they need to handle such interruption. When enabled, the FIQ are handled in monitor mode and non secure world is not allowed to mask these events. Platforms that enable FIQ handling in SP_MIN shall implement the api `sp_min_plat_fiq_handler()`. The default value is 0.
- `TRUSTED_BOARD_BOOT`: Boolean flag to include support for the Trusted Board Boot feature. When set to ‘1’, BL1 and BL2 images include support to load and verify the certificates and images in a FIP, and BL1 includes support for the Firmware Update. The default value is ‘0’. Generation and inclusion of certificates in the FIP and FWU_FIP depends upon the value of the `GENERATE_COT` option.

  Warning: This option depends on `CREATE_KEYS` to be enabled. If the keys already exist in disk, they will be overwritten without further notice.

- `TRUSTED_WORLD_KEY`: This option is used when `GENERATE_COT=1`. It specifies the file that contains the Trusted World private key in PEM format. If `SAVE_KEYS=1`, this file name will be used to save the key.
- `TSP_INIT_ASYNC`: Choose BL32 initialization method as asynchronous or synchronous, (see “Initializing a BL32 Image” section in *Firmware Design*). It can take the value 0 (BL32 is initialized using synchronous method) or 1 (BL32 is initialized using asynchronous method). Default is 0.
- `TSP_NS_INTR_ASYNC_PREEMPT`: A non zero value enables the interrupt routing model which routes non-secure interrupts asynchronously from TSP to EL3 causing immediate preemption of TSP. The EL3 is responsible for saving and restoring the TSP context in this routing model. The default routing model (when the value is 0) is to route non-secure interrupts to TSP allowing it to save its context and hand over synchronously to EL3 via an SMC.

  Note: When `EL3_EXCEPTION_HANDLING` is `1`, `TSP_NS_INTR_ASYNC_PREEMPT` must also be set to `1`.

- `USE_ARM_LINK`: This flag determines whether to enable support for ARM linker. When the `LINKER` build variable points to the armlink linker, this flag is enabled automatically. To enable support for armlink, platforms will have to provide a scatter file for the BL image. Currently, Tegra platforms use the armlink support to compile BL3-1 images.
- `USE_COHERENT_MEM`: This flag determines whether to include the coherent memory region in the BL memory map or not (see “Use of Coherent memory in TF-A” section in *Firmware Design*). It can take the value 1 (Coherent memory region is included) or 0 (Coherent memory region is excluded). Default is 1.
- `USE_DEBUGFS`: When set to 1 this option activates an EXPERIMENTAL feature exposing a virtual filesystem interface through BL31 as a SiP SMC function. Default is 0.
- `ARM_IO_IN_DTB`: This flag determines whether to use IO based on the firmware configuration framework. This will move the io_policies into a configuration device tree, instead of static structure in the code base. This

is currently an experimental feature.

- `COT_DESC_IN_DTB`: This flag determines whether to create COT descriptors at runtime using fconf. If this flag is enabled, COT descriptors are statically captured in tb_fw_config file in the form of device tree nodes and properties. Currently, COT descriptors used by BL2 are moved to the device tree and COT descriptors used by BL1 are retained in the code base statically. This is currently an experimental feature.
- `SDEI_IN_FCONF`: This flag determines whether to configure SDEI setup in runtime using firmware configuration framework. The platform specific SDEI shared and private events configuration is retrieved from device tree rather than static C structures at compile time. This is currently an experimental feature and is only supported if SDEI_SUPPORT build flag is enabled.
- `SEC_INT_DESC_IN_FCONF`: This flag determines whether to configure Group 0 and Group1 secure interrupts using the firmware configuration framework. The platform specific secure interrupt property descriptor is retrieved from device tree in runtime rather than depending on static C structure at compile time. This is currently an experimental feature.
- `USE_ROMLIB`: This flag determines whether library at ROM will be used. This feature creates a library of functions to be placed in ROM and thus reduces SRAM usage. Refer to *Library at ROM* for further details. Default is 0.
- `V`: Verbose build. If assigned anything other than 0, the build commands are printed. Default is 0.
- `VERSION_STRING`: String used in the log output for each TF-A image. Defaults to a string formed by concatenating the version number, build type and build string.
- `W`: Warning level. Some compiler warning options of interest have been regrouped and put in the root Makefile. This flag can take the values 0 to 3, each level enabling more warning options. Default is 0.
- `WARMBOOT_ENABLE_DCACHE_EARLY` : Boolean option to enable D-cache early on the CPU after warm boot. This is applicable for platforms which do not require interconnect programming to enable cache coherency (eg: single cluster platforms). If this option is enabled, then warm boot path enables D-caches immediately after enabling MMU. This option defaults to 0.
- `SUPPORT_STACK_MEMTAG`: This flag determines whether to enable memory tagging for stack or not. It accepts 2 values: `yes` and `no`. The default value of this flag is `no`. Note this option must be enabled only for ARM architecture greater than Armv8.5-A.
- `ERRATA_SPECULATIVE_AT`: This flag determines whether to enable `AT` speculative errata workaround or not. It accepts 2 values: `1` and `0`. The default value of this flag is `0`.

  `AT` speculative errata workaround disables stage1 page table walk for lower ELs (EL1 and EL0) in EL3 so that `AT` speculative fetch at any point produces either the correct result or failure without TLB allocation.

  This boolean option enables errata for all below CPUs.

Errata	CPU	Workaround Define
1165522	Cortex-A76	`ERRATA_A76_1165522`
1319367	Cortex-A72	`ERRATA_A72_1319367`
1319537	Cortex-A57	`ERRATA_A57_1319537`
1530923	Cortex-A55	`ERRATA_A55_1530923`
1530924	Cortex-A53	`ERRATA_A53_1530924`

Note: This option is enabled by build only if platform sets any of above defines mentioned in 'Workaround Define' column in the table. If this option is enabled for the EL3 software then EL2 software also must implement this workaround due to the behaviour of the errata mentioned in new SDEN document which will get published soon.

- `RAS_TRAP_LOWER_EL_ERR_ACCESS`: This flag enables/disables the SCR_EL3.TERR bit, to trap access to the RAS ERR and RAS ERX registers from lower ELs. This flag is disabled by default.
- `OPENSSL_DIR`: This flag is used to provide the installed openssl directory path on the host machine which is used to build certificate generation and firmware encryption tool.
- `USE_SP804_TIMER`: Use the SP804 timer instead of the Generic Timer for functions that wait for an arbitrary time length (udelay and mdelay). The default value is 0.

2.5.2 GICv3 driver options

GICv3 driver files are included using directive:

`include drivers/arm/gic/v3/gicv3.mk`

The driver can be configured with the following options set in the platform makefile:

- `GICV3_SUPPORT_GIC600`: Add support for the GIC-600 variants of GICv3. Enabling this option will add runtime detection support for the GIC-600, so is safe to select even for a GIC500 implementation. This option defaults to 0.
- `GICV3_IMPL_GIC600_MULTICHIP`: Selects GIC-600 variant with multichip functionality. This option defaults to 0
- `GICV3_OVERRIDE_DISTIF_PWR_OPS`: Allows override of default implementation of `arm_gicv3_distif_pre_save` and `arm_gicv3_distif_post_restore` functions. This is required for FVP platform which need to simulate GIC save and restore during SYSTEM_SUSPEND without powering down GIC. Default is 0.
- `GIC_ENABLE_V4_EXTN` : Enables GICv4 related changes in GICv3 driver. This option defaults to 0.
- `GIC_EXT_INTID`: When set to `1`, GICv3 driver will support extended PPI (1056-1119) and SPI (4096-5119) range. This option defaults to 0.

2.5.3 Debugging options

To compile a debug version and make the build more verbose use

```
make PLAT=<platform> DEBUG=1 V=1 all
```

AArch64 GCC uses DWARF version 4 debugging symbols by default. Some tools (for example DS-5) might not support this and may need an older version of DWARF symbols to be emitted by GCC. This can be achieved by using the `-gdwarf-<version>` flag, with the version being set to 2 or 3. Setting the version to 2 is recommended for DS-5 versions older than 5.16.

When debugging logic problems it might also be useful to disable all compiler optimizations by using `-O0`.

Warning: Using `-O0` could cause output images to be larger and base addresses might need to be recalculated (see the **Memory layout on Arm development platforms** section in the *Firmware Design*).

Extra debug options can be passed to the build system by setting `CFLAGS` or `LDFLAGS`:

```
CFLAGS='-O0 -gdwarf-2'                                     \
make PLAT=<platform> DEBUG=1 V=1 all
```

Note that using `-Wl,` style compilation driver options in `CFLAGS` will be ignored as the linker is called directly.

It is also possible to introduce an infinite loop to help in debugging the post-BL2 phase of TF-A. This can be done by rebuilding BL1 with the `SPIN_ON_BL1_EXIT=1` build flag. Refer to the *Common build options* section. In this case, the developer may take control of the target using a debugger when indicated by the console output. When using DS-5, the following commands can be used:

```
# Stop target execution
interrupt

#
# Prepare your debugging environment, e.g. set breakpoints
#

# Jump over the debug loop
set var $AARCH64::$Core::$PC = $AARCH64::$Core::$PC + 4

# Resume execution
continue
```

2.6 Image Terminology

This page contains the current name, abbreviated name and purpose of the various images referred to in the Trusted Firmware project.

2.6.1 General Notes

- Some of the names and abbreviated names have changed to accommodate new requirements. The changed names are as backward compatible as possible to minimize confusion. Where applicable, the previous names are indicated. Some code, documentation and build artefacts may still refer to the previous names; these will inevitably take time to catch up.
- The main name change is to prefix each image with the processor it corresponds to (for example `AP_`, `SCP_`, ...). In situations where there is no ambiguity (for example, within AP specific code/documentation), it is permitted to omit the processor prefix (for example, just BL1 instead of `AP_BL1`).
- Previously, the format for 3rd level images had 2 forms; `BL3` was either suffixed with a dash ("-") followed by a number (for example, `BL3-1`) or a subscript number, depending on whether rich text formatting was available. This was confusing and often the dash gets omitted in practice. Therefore the new form is to just omit the dash and not use subscript formatting.
- The names no longer contain dash ("-") characters at all. In some places (for example, function names) it's not possible to use this character. All dashes are either removed or replaced by underscores ("_").
- The abbreviation BL stands for BootLoader. This is a historical anomaly. Clearly, many of these images are not BootLoaders, they are simply firmware images. However, the BL abbreviation is now widely used and is retained for backwards compatibility.
- The image names are not case sensitive. For example, `bl1` is interchangeable with `BL1`, although mixed case should be avoided.

2.6.2 Trusted Firmware Images

AP Boot ROM: `AP_BL1`

Typically, this is the first code to execute on the AP and cannot be modified. Its primary purpose is to perform the minimum initialization necessary to load and authenticate an updateable AP firmware image into an executable RAM location, then hand-off control to that image.

AP RAM Firmware: `AP_BL2`

This is the 2nd stage AP firmware. It is currently also known as the "Trusted Boot Firmware". Its primary purpose is to perform any additional initialization required to load and authenticate all 3rd level firmware images into their executable RAM locations, then hand-off control to the EL3 Runtime Firmware.

EL3 Runtime Firmware: `AP_BL31`

Also known as "SoC AP firmware" or "EL3 monitor firmware". Its primary purpose is to handle transitions between the normal and secure world.

Secure-EL1 Payload (SP): `AP_BL32`

Typically this is a TEE or Trusted OS, providing runtime secure services to the normal world. However, it may refer to a more abstract Secure-EL1 Payload (SP). Note that this abbreviation should only be used in systems where there is a single or primary image executing at Secure-EL1. In systems where there are potentially multiple SPs and there is no concept of a primary SP, this abbreviation should be avoided; use the recommended **Other AP 3rd level images** abbreviation instead.

AP Normal World Firmware: `AP_BL33`

For example, UEFI or uboot. Its primary purpose is to boot a normal world OS.

Other AP 3rd level images: `AP_BL3_XXX`

The abbreviated names of the existing 3rd level images imply a load/execution ordering (for example, `AP_BL31 -> AP_BL32 -> AP_BL33`). Some systems may have additional images and/or a different load/execution ordering. The abbreviated names of the existing images are retained for backward compatibility but new 3rd level images should be suffixed with an underscore followed by text identifier, not a number.

In systems where 3rd level images are provided by different vendors, the abbreviated name should identify the vendor as well as the image function. For example, `AP_BL3_ARM_RAS`.

SCP Boot ROM: `SCP_BL1` (previously `BL0`)

Typically, this is the first code to execute on the SCP and cannot be modified. Its primary purpose is to perform the minimum initialization necessary to load and authenticate an updateable SCP firmware image into an executable RAM location, then hand-off control to that image. This may be performed in conjunction with other processor firmware (for example, `AP_BL1` and `AP_BL2`).

This image was previously abbreviated as `BL0` but in some systems, the SCP may directly load/authenticate its own firmware. In these systems, it doesn't make sense to interleave the image terminology for AP and SCP; both AP and SCP Boot ROMs are `BL1` from their own point of view.

SCP RAM Firmware: `SCP_BL2` (previously `BL3-0`)

This is the 2nd stage SCP firmware. It is currently also known as the "SCP runtime firmware" but it could potentially be an intermediate firmware if the SCP needs to load/authenticate multiple 3rd level images in future.

This image was previously abbreviated as BL3-0 but from the SCP's point of view, this has always been the 2nd stage firmware. The previous name is too AP-centric.

2.6.3 Firmware Update (FWU) Images

The terminology for these images has not been widely adopted yet but they have to be considered in a production Trusted Board Boot solution.

AP Firmware Update Boot ROM: `AP_NS_BL1U`

Typically, this is the first normal world code to execute on the AP during a firmware update operation, and cannot be modified. Its primary purpose is to load subsequent firmware update images from an external interface and communicate with `AP_BL1` to authenticate those images.

During firmware update, there are (potentially) multiple transitions between the secure and normal world. The "level" of the BL image is relative to the world it's in so it makes sense to encode "NS" in the normal world images. The absence of "NS" implies a secure world image.

AP Firmware Update Config: `AP_BL2U`

This image does the minimum necessary AP secure world configuration required to complete the firmware update operation. It is potentially a subset of `AP_BL2` functionality.

SCP Firmware Update Config: `SCP_BL2U` (previously `BL2-U0`)

This image does the minimum necessary SCP secure world configuration required to complete the firmware update operation. It is potentially a subset of `SCP_BL2` functionality.

AP Firmware Updater: `AP_NS_BL2U` (previously `BL3-U`)

This is the 2nd stage AP normal world firmware updater. Its primary purpose is to load a new set of firmware images from an external interface and write them into non-volatile storage.

2.6.4 Other Processor Firmware Images

Some systems may have additional processors to the AP and SCP. For example, a Management Control Processor (MCP). Images for these processors should follow the same terminology, with the processor abbreviation prefix, followed by underscore and the level of the firmware image.

For example,

MCP Boot ROM: `MCP_BL1`

MCP RAM Firmware: `MCP_BL2`

2.7 Porting Guide

2.7.1 Introduction

Porting Trusted Firmware-A (TF-A) to a new platform involves making some mandatory and optional modifications for both the cold and warm boot paths. Modifications consist of:

- Implementing a platform-specific function or variable,
- Setting up the execution context in a certain way, or
- Defining certain constants (for example #defines).

The platform-specific functions and variables are declared in `include/plat/common/platform.h`. The firmware provides a default implementation of variables and functions to fulfill the optional requirements. These implementations are all weakly defined; they are provided to ease the porting effort. Each platform port can override them with its own implementation if the default implementation is inadequate.

Some modifications are common to all Boot Loader (BL) stages. Section 2 discusses these in detail. The subsequent sections discuss the remaining modifications for each BL stage in detail.

Please refer to the *Platform Compatibility Policy* for the policy regarding compatibility and deprecation of these porting interfaces.

Only Arm development platforms (such as FVP and Juno) may use the functions/definitions in `include/plat/arm/common/` and the corresponding source files in `plat/arm/common/`. This is done so that there are no dependencies between platforms maintained by different people/companies. If you want to use any of the functionality present in `plat/arm` files, please create a pull request that moves the code to `plat/common` so that it can be discussed.

2.7.2 Common modifications

This section covers the modifications that should be made by the platform for each BL stage to correctly port the firmware stack. They are categorized as either mandatory or optional.

2.7.3 Common mandatory modifications

A platform port must enable the Memory Management Unit (MMU) as well as the instruction and data caches for each BL stage. Setting up the translation tables is the responsibility of the platform port because memory maps differ across platforms. A memory translation library (see `lib/xlat_tables/`) is provided to help in this setup.

Note that although this library supports non-identity mappings, this is intended only for re-mapping peripheral physical addresses and allows platforms with high I/O addresses to reduce their virtual address space. All other addresses corresponding to code and data must currently use an identity mapping.

Also, the only translation granule size supported in TF-A is 4KB, as various parts of the code assume that is the case. It is not possible to switch to 16 KB or 64 KB granule sizes at the moment.

In Arm standard platforms, each BL stage configures the MMU in the platform-specific architecture setup function, `blX_plat_arch_setup()`, and uses an identity mapping for all addresses.

If the build option `USE_COHERENT_MEM` is enabled, each platform can allocate a block of identity mapped secure memory with Device-nGnRE attributes aligned to page boundary (4K) for each BL stage. All sections which allocate coherent memory are grouped under `coherent_ram`. For ex: Bakery locks are placed in a section identified by name `bakery_lock` inside `coherent_ram` so that its possible for the firmware to place variables in it using the following C code directive:

```
__section("bakery_lock")
```

Or alternatively the following assembler code directive:

```
.section bakery_lock
```

The `coherent_ram` section is a sum of all sections like `bakery_lock` which are used to allocate any data structures that are accessed both when a CPU is executing with its MMU and caches enabled, and when it's running with its MMU and caches disabled. Examples are given below.

The following variables, functions and constants must be defined by the platform for the firmware to work correctly.

File : platform_def.h [mandatory]

Each platform must ensure that a header file of this name is in the system include path with the following constants defined. This will require updating the list of `PLAT_INCLUDES` in the `platform.mk` file.

Platform ports may optionally use the file `include/plat/common/common_def.h`, which provides typical values for some of the constants below. These values are likely to be suitable for all platform ports.

- **#define : PLATFORM_LINKER_FORMAT**

  Defines the linker format used by the platform, for example `elf64-littleaarch64`.

- **#define : PLATFORM_LINKER_ARCH**

  Defines the processor architecture for the linker by the platform, for example `aarch64`.

- **#define : PLATFORM_STACK_SIZE**

  Defines the normal stack memory available to each CPU. This constant is used by `plat/common/aarch64/platform_mp_stack.S` and `plat/common/aarch64/platform_up_stack.S`.

- **define : CACHE_WRITEBACK_GRANULE**

  Defines the size in bits of the largest cache line across all the cache levels in the platform.

- **#define : FIRMWARE_WELCOME_STR**

  Defines the character string printed by BL1 upon entry into the `bl1_main()` function.

- **#define : PLATFORM_CORE_COUNT**

  Defines the total number of CPUs implemented by the platform across all clusters in the system.

- **#define : PLAT_NUM_PWR_DOMAINS**

  Defines the total number of nodes in the power domain topology tree at all the power domain levels used by the platform. This macro is used by the PSCI implementation to allocate data structures to represent power domain topology.

- **#define : PLAT_MAX_PWR_LVL**

  Defines the maximum power domain level that the power management operations should apply to. More often, but not always, the power domain level corresponds to affinity level. This macro allows the PSCI implementation to know the highest power domain level that it should consider for power management operations in the system that the platform implements. For example, the Base AEM FVP implements two clusters with a configurable number of CPUs and it reports the maximum power domain level as 1.

- **#define : PLAT_MAX_OFF_STATE**

  Defines the local power state corresponding to the deepest power down possible at every power domain level in the platform. The local power states for each level may be sparsely allocated between 0 and this value with 0 being reserved for the RUN state. The PSCI implementation uses this value to initialize the local power states of the power domain nodes and to specify the requested power state for a PSCI_CPU_OFF call.

- **#define : PLAT_MAX_RET_STATE**

  Defines the local power state corresponding to the deepest retention state possible at every power domain level in the platform. This macro should be a value less than PLAT_MAX_OFF_STATE and greater than 0. It is used by the PSCI implementation to distinguish between retention and power down local power states within PSCI_CPU_SUSPEND call.

- **#define : PLAT_MAX_PWR_LVL_STATES**

  Defines the maximum number of local power states per power domain level that the platform supports. The default value of this macro is 2 since most platforms just support a maximum of two local power states at each power domain level (power-down and retention). If the platform needs to account for more local power states, then it must redefine this macro.

  Currently, this macro is used by the Generic PSCI implementation to size the array used for PSCI_STAT_COUNT/RESIDENCY accounting.

- **#define : BL1_RO_BASE**

  Defines the base address in secure ROM where BL1 originally lives. Must be aligned on a page-size boundary.

- **#define : BL1_RO_LIMIT**

  Defines the maximum address in secure ROM that BL1's actual content (i.e. excluding any data section allocated at runtime) can occupy.

- **#define : BL1_RW_BASE**

  Defines the base address in secure RAM where BL1's read-write data will live at runtime. Must be aligned on a page-size boundary.

- **#define : BL1_RW_LIMIT**

  Defines the maximum address in secure RAM that BL1's read-write data can occupy at runtime.

- **#define : BL2_BASE**

  Defines the base address in secure RAM where BL1 loads the BL2 binary image. Must be aligned on a page-size boundary. This constant is not applicable when BL2_IN_XIP_MEM is set to '1'.

- **#define : BL2_LIMIT**

  Defines the maximum address in secure RAM that the BL2 image can occupy. This constant is not applicable when BL2_IN_XIP_MEM is set to '1'.

- **#define : BL2_RO_BASE**

  Defines the base address in secure XIP memory where BL2 RO section originally lives. Must be aligned on a page-size boundary. This constant is only needed when BL2_IN_XIP_MEM is set to '1'.

- **#define : BL2_RO_LIMIT**

  Defines the maximum address in secure XIP memory that BL2's actual content (i.e. excluding any data section allocated at runtime) can occupy. This constant is only needed when BL2_IN_XIP_MEM is set to '1'.

- **#define : BL2_RW_BASE**

  Defines the base address in secure RAM where BL2's read-write data will live at runtime. Must be aligned on a page-size boundary. This constant is only needed when BL2_IN_XIP_MEM is set to '1'.

- **#define : BL2_RW_LIMIT**

  Defines the maximum address in secure RAM that BL2's read-write data can occupy at runtime. This constant is only needed when BL2_IN_XIP_MEM is set to '1'.

- **#define : BL31_BASE**

  Defines the base address in secure RAM where BL2 loads the BL31 binary image. Must be aligned on a page-size boundary.

- **#define : BL31_LIMIT**

  Defines the maximum address in secure RAM that the BL31 image can occupy.

For every image, the platform must define individual identifiers that will be used by BL1 or BL2 to load the corresponding image into memory from non-volatile storage. For the sake of performance, integer numbers will be used as identifiers. The platform will use those identifiers to return the relevant information about the image to be loaded (file handler, load address, authentication information, etc.). The following image identifiers are mandatory:

- **#define : BL2_IMAGE_ID**

  BL2 image identifier, used by BL1 to load BL2.

- **#define : BL31_IMAGE_ID**

  BL31 image identifier, used by BL2 to load BL31.

- **#define : BL33_IMAGE_ID**

  BL33 image identifier, used by BL2 to load BL33.

If Trusted Board Boot is enabled, the following certificate identifiers must also be defined:

- **#define : TRUSTED_BOOT_FW_CERT_ID**

  BL2 content certificate identifier, used by BL1 to load the BL2 content certificate.

- **#define : TRUSTED_KEY_CERT_ID**

  Trusted key certificate identifier, used by BL2 to load the trusted key certificate.

- **#define : SOC_FW_KEY_CERT_ID**

  BL31 key certificate identifier, used by BL2 to load the BL31 key certificate.

- **#define : SOC_FW_CONTENT_CERT_ID**

  BL31 content certificate identifier, used by BL2 to load the BL31 content certificate.

- **#define : NON_TRUSTED_FW_KEY_CERT_ID**

  BL33 key certificate identifier, used by BL2 to load the BL33 key certificate.

- **#define : NON_TRUSTED_FW_CONTENT_CERT_ID**

  BL33 content certificate identifier, used by BL2 to load the BL33 content certificate.

- **#define : FWU_CERT_ID**

  Firmware Update (FWU) certificate identifier, used by NS_BL1U to load the FWU content certificate.

- **#define : PLAT_CRYPTOCELL_BASE**

  This defines the base address of Arm® TrustZone® CryptoCell and must be defined if CryptoCell crypto driver is used for Trusted Board Boot. For capable Arm platforms, this driver is used if `ARM_CRYPTOCELL_INTEG` is set.

If the AP Firmware Updater Configuration image, BL2U is used, the following must also be defined:

- **#define : BL2U_BASE**

  Defines the base address in secure memory where BL1 copies the BL2U binary image. Must be aligned on a page-size boundary.

- **#define : BL2U_LIMIT**

  Defines the maximum address in secure memory that the BL2U image can occupy.

- **#define : BL2U_IMAGE_ID**

  BL2U image identifier, used by BL1 to fetch an image descriptor corresponding to BL2U.

If the SCP Firmware Update Configuration Image, SCP_BL2U is used, the following must also be defined:

- **#define : SCP_BL2U_IMAGE_ID**

  SCP_BL2U image identifier, used by BL1 to fetch an image descriptor corresponding to SCP_BL2U.

  Note: TF-A does not provide source code for this image.

If the Non-Secure Firmware Updater ROM, NS_BL1U is used, the following must also be defined:

- **#define : NS_BL1U_BASE**

  Defines the base address in non-secure ROM where NS_BL1U executes. Must be aligned on a page-size boundary.

  Note: TF-A does not provide source code for this image.

- **#define : NS_BL1U_IMAGE_ID**

  NS_BL1U image identifier, used by BL1 to fetch an image descriptor corresponding to NS_BL1U.

If the Non-Secure Firmware Updater, NS_BL2U is used, the following must also be defined:

- **#define : NS_BL2U_BASE**

  Defines the base address in non-secure memory where NS_BL2U executes. Must be aligned on a page-size boundary.

  Note: TF-A does not provide source code for this image.

- **#define : NS_BL2U_IMAGE_ID**

  NS_BL2U image identifier, used by BL1 to fetch an image descriptor corresponding to NS_BL2U.

For the the Firmware update capability of TRUSTED BOARD BOOT, the following macros may also be defined:

- **#define : PLAT_FWU_MAX_SIMULTANEOUS_IMAGES**

  Total number of images that can be loaded simultaneously. If the platform doesn't specify any value, it defaults to 10.

If a SCP_BL2 image is supported by the platform, the following constants must also be defined:

- **#define : SCP_BL2_IMAGE_ID**

  SCP_BL2 image identifier, used by BL2 to load SCP_BL2 into secure memory from platform storage before being transferred to the SCP.

- **#define : SCP_FW_KEY_CERT_ID**

  SCP_BL2 key certificate identifier, used by BL2 to load the SCP_BL2 key certificate (mandatory when Trusted Board Boot is enabled).

- **#define : SCP_FW_CONTENT_CERT_ID**

  SCP_BL2 content certificate identifier, used by BL2 to load the SCP_BL2 content certificate (mandatory when Trusted Board Boot is enabled).

If a BL32 image is supported by the platform, the following constants must also be defined:

- **#define : BL32_IMAGE_ID**

  BL32 image identifier, used by BL2 to load BL32.

- **#define : TRUSTED_OS_FW_KEY_CERT_ID**

  BL32 key certificate identifier, used by BL2 to load the BL32 key certificate (mandatory when Trusted Board Boot is enabled).

- **#define : TRUSTED_OS_FW_CONTENT_CERT_ID**

  BL32 content certificate identifier, used by BL2 to load the BL32 content certificate (mandatory when Trusted Board Boot is enabled).

- **#define : BL32_BASE**

  Defines the base address in secure memory where BL2 loads the BL32 binary image. Must be aligned on a page-size boundary.

- **#define : BL32_LIMIT**

  Defines the maximum address that the BL32 image can occupy.

If the Test Secure-EL1 Payload (TSP) instantiation of BL32 is supported by the platform, the following constants must also be defined:

- **#define : TSP_SEC_MEM_BASE**

  Defines the base address of the secure memory used by the TSP image on the platform. This must be at the same address or below `BL32_BASE`.

- **#define : TSP_SEC_MEM_SIZE**

  Defines the size of the secure memory used by the BL32 image on the platform. `TSP_SEC_MEM_BASE` and `TSP_SEC_MEM_SIZE` must fully accommodate the memory required by the BL32 image, defined by `BL32_BASE` and `BL32_LIMIT`.

- **#define : TSP_IRQ_SEC_PHY_TIMER**

  Defines the ID of the secure physical generic timer interrupt used by the TSP's interrupt handling code.

If the platform port uses the translation table library code, the following constants must also be defined:

- **#define : PLAT_XLAT_TABLES_DYNAMIC**

  Optional flag that can be set per-image to enable the dynamic allocation of regions even when the MMU is enabled. If not defined, only static functionality will be available, if defined and set to 1 it will also include the dynamic functionality.

- **#define : MAX_XLAT_TABLES**

  Defines the maximum number of translation tables that are allocated by the translation table library code. To minimize the amount of runtime memory used, choose the smallest value needed to map the required virtual addresses for each BL stage. If `PLAT_XLAT_TABLES_DYNAMIC` flag is enabled for a BL image, `MAX_XLAT_TABLES` must be defined to accommodate the dynamic regions as well.

- **#define : MAX_MMAP_REGIONS**

  Defines the maximum number of regions that are allocated by the translation table library code. A region consists of physical base address, virtual base address, size and attributes (Device/Memory, RO/RW, Secure/Non-Secure), as defined in the `mmap_region_t` structure. The platform defines the regions that should be mapped. Then, the translation table library will create the corresponding tables and descriptors at runtime. To minimize the amount of runtime memory used, choose the smallest value needed to register the required regions for each BL stage. If `PLAT_XLAT_TABLES_DYNAMIC` flag is enabled for a BL image, `MAX_MMAP_REGIONS` must be defined to accommodate the dynamic regions as well.

- **#define : PLAT_VIRT_ADDR_SPACE_SIZE**

  Defines the total size of the virtual address space in bytes. For example, for a 32 bit virtual address space, this value should be `(1ULL << 32)`.

- **#define : PLAT_PHY_ADDR_SPACE_SIZE**

  Defines the total size of the physical address space in bytes. For example, for a 32 bit physical address space, this value should be `(1ULL << 32)`.

If the platform port uses the IO storage framework, the following constants must also be defined:

- **#define : MAX_IO_DEVICES**

  Defines the maximum number of registered IO devices. Attempting to register more devices than this value using `io_register_device()` will fail with -ENOMEM.

- **#define : MAX_IO_HANDLES**

  Defines the maximum number of open IO handles. Attempting to open more IO entities than this value using `io_open()` will fail with -ENOMEM.

- **#define : MAX_IO_BLOCK_DEVICES**

Defines the maximum number of registered IO block devices. Attempting to register more devices this value using `io_dev_open()` will fail with -ENOMEM. MAX_IO_BLOCK_DEVICES should be less than MAX_IO_DEVICES. With this macro, multiple block devices could be supported at the same time.

If the platform needs to allocate data within the per-cpu data framework in BL31, it should define the following macro. Currently this is only required if the platform decides not to use the coherent memory section by undefining the `USE_COHERENT_MEM` build flag. In this case, the framework allocates the required memory within the the per-cpu data to minimize wastage.

- **#define : PLAT_PCPU_DATA_SIZE**

  Defines the memory (in bytes) to be reserved within the per-cpu data structure for use by the platform layer.

The following constants are optional. They should be defined when the platform memory layout implies some image overlaying like in Arm standard platforms.

- **#define : BL31_PROGBITS_LIMIT**

  Defines the maximum address in secure RAM that the BL31's progbits sections can occupy.

- **#define : TSP_PROGBITS_LIMIT**

  Defines the maximum address that the TSP's progbits sections can occupy.

If the platform port uses the PL061 GPIO driver, the following constant may optionally be defined:

- **PLAT_PL061_MAX_GPIOS** Maximum number of GPIOs required by the platform. This allows control how much memory is allocated for PL061 GPIO controllers. The default value is

  1. $(eval $(call add_define,PLAT_PL061_MAX_GPIOS))

If the platform port uses the partition driver, the following constant may optionally be defined:

- **PLAT_PARTITION_MAX_ENTRIES** Maximum number of partition entries required by the platform. This allows control how much memory is allocated for partition entries. The default value is 128. For example, define the build flag in `platform.mk`: PLAT_PARTITION_MAX_ENTRIES := 12 $(eval $(call add_define,PLAT_PARTITION_MAX_ENTRIES))
- **PLAT_PARTITION_BLOCK_SIZE** The size of partition block. It could be either 512 bytes or 4096 bytes. The default value is 512. For example, define the build flag in `platform.mk`: PLAT_PARTITION_BLOCK_SIZE := 4096 $(eval $(call add_define,PLAT_PARTITION_BLOCK_SIZE))

The following constant is optional. It should be defined to override the default behaviour of the `assert()` function (for example, to save memory).

- **PLAT_LOG_LEVEL_ASSERT** If `PLAT_LOG_LEVEL_ASSERT` is higher or equal than `LOG_LEVEL_VERBOSE`, `assert()` prints the name of the file, the line number and the asserted expression. Else if it is higher than `LOG_LEVEL_INFO`, it prints the file name and the line number. Else if it is lower than `LOG_LEVEL_INFO`, it doesn't print anything to the console. If `PLAT_LOG_LEVEL_ASSERT` isn't defined, it defaults to `LOG_LEVEL`.

If the platform port uses the Activity Monitor Unit, the following constant may be defined:

- **PLAT_AMU_GROUP1_COUNTERS_MASK** This mask reflects the set of group counters that should be enabled. The maximum number of group 1 counters supported by AMUv1 is 16 so the mask can be at most 0xffff. If the platform does not define this mask, no group 1 counters are enabled.

File : plat_macros.S [mandatory]

Each platform must ensure a file of this name is in the system include path with the following macro defined. In the Arm development platforms, this file is found in `plat/arm/board/<plat_name>/include/plat_macros.S`.

- **Macro : plat_crash_print_regs**

  This macro allows the crash reporting routine to print relevant platform registers in case of an unhandled exception in BL31. This aids in debugging and this macro can be defined to be empty in case register reporting is not desired.

  For instance, GIC or interconnect registers may be helpful for troubleshooting.

2.7.4 Handling Reset

BL1 by default implements the reset vector where execution starts from a cold or warm boot. BL31 can be optionally set as a reset vector using the `RESET_TO_BL31` make variable.

For each CPU, the reset vector code is responsible for the following tasks:

1. Distinguishing between a cold boot and a warm boot.
2. In the case of a cold boot and the CPU being a secondary CPU, ensuring that the CPU is placed in a platform-specific state until the primary CPU performs the necessary steps to remove it from this state.
3. In the case of a warm boot, ensuring that the CPU jumps to a platform- specific address in the BL31 image in the same processor mode as it was when released from reset.

The following functions need to be implemented by the platform port to enable reset vector code to perform the above tasks.

Function : plat_get_my_entrypoint() [mandatory when PROGRAMMABLE_RESET_ADDRESS == 0]

```
Argument : void
Return   : uintptr_t
```

This function is called with the MMU and caches disabled (`SCTLR_EL3.M` = 0 and `SCTLR_EL3.C` = 0). The function is responsible for distinguishing between a warm and cold reset for the current CPU using platform-specific means. If it's a warm reset, then it returns the warm reset entrypoint point provided to `plat_setup_psci_ops()` during BL31 initialization. If it's a cold reset then this function must return zero.

This function does not follow the Procedure Call Standard used by the Application Binary Interface for the Arm 64-bit architecture. The caller should not assume that callee saved registers are preserved across a call to this function.

This function fulfills requirement 1 and 3 listed above.

Note that for platforms that support programming the reset address, it is expected that a CPU will start executing code directly at the right address, both on a cold and warm reset. In this case, there is no need to identify the type of reset nor to query the warm reset entrypoint. Therefore, implementing this function is not required on such platforms.

Function : plat_secondary_cold_boot_setup() [mandatory when COLD_BOOT_SINGLE_CPU == 0]

```
Argument : void
```

This function is called with the MMU and data caches disabled. It is responsible for placing the executing secondary CPU in a platform-specific state until the primary CPU performs the necessary actions to bring it out of that state and allow entry into the OS. This function must not return.

In the Arm FVP port, when using the normal boot flow, each secondary CPU powers itself off. The primary CPU is responsible for powering up the secondary CPUs when normal world software requires them. When booting an EL3 payload instead, they stay powered on and are put in a holding pen until their mailbox gets populated.

This function fulfills requirement 2 above.

Note that for platforms that can't release secondary CPUs out of reset, only the primary CPU will execute the cold boot code. Therefore, implementing this function is not required on such platforms.

Function : plat_is_my_cpu_primary() [mandatory when COLD_BOOT_SINGLE_CPU == 0]

```
Argument : void
Return   : unsigned int
```

This function identifies whether the current CPU is the primary CPU or a secondary CPU. A return value of zero indicates that the CPU is not the primary CPU, while a non-zero return value indicates that the CPU is the primary CPU.

Note that for platforms that can't release secondary CPUs out of reset, only the primary CPU will execute the cold boot code. Therefore, there is no need to distinguish between primary and secondary CPUs and implementing this function is not required.

Function : platform_mem_init() [mandatory]

```
Argument : void
Return   : void
```

This function is called before any access to data is made by the firmware, in order to carry out any essential memory initialization.

Function: plat_get_rotpk_info()

```
Argument : void *, void **, unsigned int *, unsigned int *
Return   : int
```

This function is mandatory when Trusted Board Boot is enabled. It returns a pointer to the ROTPK stored in the platform (or a hash of it) and its length. The ROTPK must be encoded in DER format according to the following ASN.1 structure:

```
AlgorithmIdentifier  ::=  SEQUENCE  {
    algorithm         OBJECT IDENTIFIER,
    parameters        ANY DEFINED BY algorithm OPTIONAL
}

SubjectPublicKeyInfo  ::=  SEQUENCE  {
```

(continues on next page)

(continued from previous page)

```
    algorithm         AlgorithmIdentifier,
    subjectPublicKey  BIT STRING
}
```

In case the function returns a hash of the key:

```
DigestInfo ::= SEQUENCE {
    digestAlgorithm   AlgorithmIdentifier,
    digest            OCTET STRING
}
```

The function returns 0 on success. Any other value is treated as error by the Trusted Board Boot. The function also reports extra information related to the ROTPK in the flags parameter:

```
ROTPK_IS_HASH      : Indicates that the ROTPK returned by the platform is a
                     hash.
ROTPK_NOT_DEPLOYED : This allows the platform to skip certificate ROTPK
                     verification while the platform ROTPK is not deployed.
                     When this flag is set, the function does not need to
                     return a platform ROTPK, and the authentication
                     framework uses the ROTPK in the certificate without
                     verifying it against the platform value. This flag
                     must not be used in a deployed production environment.
```

Function: plat_get_nv_ctr()

```
Argument : void *, unsigned int *
Return   : int
```

This function is mandatory when Trusted Board Boot is enabled. It returns the non-volatile counter value stored in the platform in the second argument. The cookie in the first argument may be used to select the counter in case the platform provides more than one (for example, on platforms that use the default TBBR CoT, the cookie will correspond to the OID values defined in TRUSTED_FW_NVCOUNTER_OID or NON_TRUSTED_FW_NVCOUNTER_OID).

The function returns 0 on success. Any other value means the counter value could not be retrieved from the platform.

Function: plat_set_nv_ctr()

```
Argument : void *, unsigned int
Return   : int
```

This function is mandatory when Trusted Board Boot is enabled. It sets a new counter value in the platform. The cookie in the first argument may be used to select the counter (as explained in plat_get_nv_ctr()). The second argument is the updated counter value to be written to the NV counter.

The function returns 0 on success. Any other value means the counter value could not be updated.

Function: plat_set_nv_ctr2()

```
Argument : void *, const auth_img_desc_t *, unsigned int
Return   : int
```

This function is optional when Trusted Board Boot is enabled. If this interface is defined, then `plat_set_nv_ctr()` need not be defined. The first argument passed is a cookie and is typically used to differentiate between a Non Trusted NV Counter and a Trusted NV Counter. The second argument is a pointer to an authentication image descriptor and may be used to decide if the counter is allowed to be updated or not. The third argument is the updated counter value to be written to the NV counter.

The function returns 0 on success. Any other value means the counter value either could not be updated or the authentication image descriptor indicates that it is not allowed to be updated.

2.7.5 Common mandatory function modifications

The following functions are mandatory functions which need to be implemented by the platform port.

Function : plat_my_core_pos()

```
Argument : void
Return   : unsigned int
```

This function returns the index of the calling CPU which is used as a CPU-specific linear index into blocks of memory (for example while allocating per-CPU stacks). This function will be invoked very early in the initialization sequence which mandates that this function should be implemented in assembly and should not rely on the availability of a C runtime environment. This function can clobber x0 - x8 and must preserve x9 - x29.

This function plays a crucial role in the power domain topology framework in PSCI and details of this can be found in *PSCI Power Domain Tree Structure*.

Function : plat_core_pos_by_mpidr()

```
Argument : u_register_t
Return   : int
```

This function validates the `MPIDR` of a CPU and converts it to an index, which can be used as a CPU-specific linear index into blocks of memory. In case the `MPIDR` is invalid, this function returns -1. This function will only be invoked by BL31 after the power domain topology is initialized and can utilize the C runtime environment. For further details about how TF-A represents the power domain topology and how this relates to the linear CPU index, please refer *PSCI Power Domain Tree Structure*.

Function : plat_get_mbedtls_heap() [when TRUSTED_BOARD_BOOT == 1]

```
Arguments : void **heap_addr, size_t *heap_size
Return    : int
```

This function is invoked during Mbed TLS library initialisation to get a heap, by means of a starting address and a size. This heap will then be used internally by the Mbed TLS library. Hence, each BL stage that utilises Mbed TLS must be able to provide a heap to it.

A helper function can be found in *drivers/auth/mbedtls/mbedtls_common.c* in which a heap is statically reserved during compile time inside every image (i.e. every BL stage) that utilises Mbed TLS. In this default implementation, the function simply returns the address and size of this "pre-allocated" heap. For a platform to use this default implementation, only a call to the helper from inside plat_get_mbedtls_heap() body is enough and nothing else is needed.

However, by writting their own implementation, platforms have the potential to optimise memory usage. For example, on some Arm platforms, the Mbed TLS heap is shared between BL1 and BL2 stages and, thus, the necessary space is not reserved twice.

On success the function should return 0 and a negative error code otherwise.

Function : plat_get_enc_key_info() [when FW_ENC_STATUS == 0 or 1]

```
Arguments : enum fw_enc_status_t fw_enc_status, uint8_t *key,
            size_t *key_len, unsigned int *flags, const uint8_t *img_id,
            size_t img_id_len
Return    : int
```

This function provides a symmetric key (either SSK or BSSK depending on fw_enc_status) which is invoked during runtime decryption of encrypted firmware images. *plat/common/plat_bl_common.c* provides a dummy weak implementation for testing purposes which must be overridden by the platform trying to implement a real world firmware encryption use-case.

It also allows the platform to pass symmetric key identifier rather than actual symmetric key which is useful in cases where the crypto backend provides secure storage for the symmetric key. So in this case `ENC_KEY_IS_IDENTIFIER` flag must be set in `flags`.

In addition to above a platform may also choose to provide an image specific symmetric key/identifier using img_id.

On success the function should return 0 and a negative error code otherwise.

Note that this API depends on `DECRYPTION_SUPPORT` build flag which is marked as experimental.

2.7.6 Common optional modifications

The following are helper functions implemented by the firmware that perform common platform-specific tasks. A platform may choose to override these definitions.

Function : plat_set_my_stack()

```
Argument : void
Return   : void
```

This function sets the current stack pointer to the normal memory stack that has been allocated for the current CPU. For BL images that only require a stack for the primary CPU, the UP version of the function is used. The size of the stack allocated to each CPU is specified by the platform defined constant `PLATFORM_STACK_SIZE`.

Common implementations of this function for the UP and MP BL images are provided in `plat/common/aarch64/platform_up_stack.S` and `plat/common/aarch64/platform_mp_stack.S`

Function : plat_get_my_stack()

```
Argument : void
Return   : uintptr_t
```

This function returns the base address of the normal memory stack that has been allocated for the current CPU. For BL images that only require a stack for the primary CPU, the UP version of the function is used. The size of the stack allocated to each CPU is specified by the platform defined constant `PLATFORM_STACK_SIZE`.

Common implementations of this function for the UP and MP BL images are provided in `plat/common/aarch64/platform_up_stack.S` and `plat/common/aarch64/platform_mp_stack.S`

Function : plat_report_exception()

```
Argument : unsigned int
Return   : void
```

A platform may need to report various information about its status when an exception is taken, for example the current exception level, the CPU security state (secure/non-secure), the exception type, and so on. This function is called in the following circumstances:

- In BL1, whenever an exception is taken.
- In BL2, whenever an exception is taken.

The default implementation doesn't do anything, to avoid making assumptions about the way the platform displays its status information.

For AArch64, this function receives the exception type as its argument. Possible values for exceptions types are listed in the `include/common/bl_common.h` header file. Note that these constants are not related to any architectural exception code; they are just a TF-A convention.

For AArch32, this function receives the exception mode as its argument. Possible values for exception modes are listed in the `include/lib/aarch32/arch.h` header file.

Function : plat_reset_handler()

```
Argument : void
Return   : void
```

A platform may need to do additional initialization after reset. This function allows the platform to do the platform specific initializations. Platform specific errata workarounds could also be implemented here. The API should preserve the values of callee saved registers x19 to x29.

The default implementation doesn't do anything. If a platform needs to override the default implementation, refer to the *Firmware Design* for general guidelines.

Function : plat_disable_acp()

```
Argument : void
Return   : void
```

This API allows a platform to disable the Accelerator Coherency Port (if present) during a cluster power down sequence. The default weak implementation doesn't do anything. Since this API is called during the power down sequence, it has restrictions for stack usage and it can use the registers x0 - x17 as scratch registers. It should preserve the value in x18 register as it is used by the caller to store the return address.

Function : plat_error_handler()

```
Argument : int
Return   : void
```

This API is called when the generic code encounters an error situation from which it cannot continue. It allows the platform to perform error reporting or recovery actions (for example, reset the system). This function must not return.

The parameter indicates the type of error using standard codes from `errno.h`. Possible errors reported by the generic code are:

- `-EAUTH`: a certificate or image could not be authenticated (when Trusted Board Boot is enabled)
- `-ENOENT`: the requested image or certificate could not be found or an IO error was detected
- `-ENOMEM`: resources exhausted. TF-A does not use dynamic memory, so this error is usually an indication of an incorrect array size

The default implementation simply spins.

Function : plat_panic_handler()

```
Argument : void
Return   : void
```

This API is called when the generic code encounters an unexpected error situation from which it cannot recover. This function must not return, and must be implemented in assembly because it may be called before the C environment is initialized.

Note: The address from where it was called is stored in x30 (Link Register). The default implementation simply spins.

Function : plat_get_bl_image_load_info()

```
Argument : void
Return   : bl_load_info_t *
```

This function returns pointer to the list of images that the platform has populated to load. This function is invoked in BL2 to load the BL3xx images.

Function : plat_get_next_bl_params()

```
Argument : void
Return   : bl_params_t *
```

This function returns a pointer to the shared memory that the platform has kept aside to pass TF-A related information that next BL image needs. This function is invoked in BL2 to pass this information to the next BL image.

Function : plat_get_stack_protector_canary()

```
Argument : void
Return   : u_register_t
```

This function returns a random value that is used to initialize the canary used when the stack protector is enabled with ENABLE_STACK_PROTECTOR. A predictable value will weaken the protection as the attacker could easily write the right value as part of the attack most of the time. Therefore, it should return a true random number.

Warning: For the protection to be effective, the global data need to be placed at a lower address than the stack bases. Failure to do so would allow an attacker to overwrite the canary as part of the stack buffer overflow attack.

Function : plat_flush_next_bl_params()

```
Argument : void
Return   : void
```

This function flushes to main memory all the image params that are passed to next image. This function is invoked in BL2 to flush this information to the next BL image.

Function : plat_log_get_prefix()

```
Argument : unsigned int
Return   : const char *
```

This function defines the prefix string corresponding to the *log_level* to be prepended to all the log output from TF-A. The *log_level* (argument) will correspond to one of the standard log levels defined in debug.h. The platform can override the common implementation to define a different prefix string for the log output. The implementation should be robust to future changes that increase the number of log levels.

Function : plat_get_soc_version()

```
Argument : void
Return   : int32_t
```

This function returns soc version which mainly consist of below fields

```
soc_version[30:24]  = JEP-106 continuation code for the SiP
soc_version[23:16]  = JEP-106 identification code with parity bit for the SiP
soc_version[15:0]   = Implementation defined SoC ID
```

Function : plat_get_soc_revision()

```
Argument : void
Return   : int32_t
```

This function returns soc revision in below format

```
soc_revision[0:30] = SOC revision of specific SOC
```

Function : plat_is_smccc_feature_available()

```
Argument : u_register_t
Return   : int32_t
```

This function returns SMC_ARCH_CALL_SUCCESS if the platform supports the SMCCC function specified in the argument; otherwise returns SMC_ARCH_CALL_NOT_SUPPORTED.

2.7.7 Modifications specific to a Boot Loader stage

2.7.8 Boot Loader Stage 1 (BL1)

BL1 implements the reset vector where execution starts from after a cold or warm boot. For each CPU, BL1 is responsible for the following tasks:

1. Handling the reset as described in section 2.2
2. In the case of a cold boot and the CPU being the primary CPU, ensuring that only this CPU executes the remaining BL1 code, including loading and passing control to the BL2 stage.
3. Identifying and starting the Firmware Update process (if required).
4. Loading the BL2 image from non-volatile storage into secure memory at the address specified by the platform defined constant `BL2_BASE`.
5. Populating a `meminfo` structure with the following information in memory, accessible by BL2 immediately upon entry.

   ```
   meminfo.total_base = Base address of secure RAM visible to BL2
   meminfo.total_size = Size of secure RAM visible to BL2
   ```

   By default, BL1 places this `meminfo` structure at the end of secure memory visible to BL2.

   It is possible for the platform to decide where it wants to place the `meminfo` structure for BL2 or restrict the amount of memory visible to BL2 by overriding the weak default implementation of `bl1_plat_handle_post_image_load` API.

The following functions need to be implemented by the platform port to enable BL1 to perform the above tasks.

Function : bl1_early_platform_setup() [mandatory]

```
Argument : void
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU.

On Arm standard platforms, this function:

- Enables a secure instance of SP805 to act as the Trusted Watchdog.
- Initializes a UART (PL011 console), which enables access to the `printf` family of functions in BL1.
- Enables issuing of snoop and DVM (Distributed Virtual Memory) requests to the CCI slave interface corresponding to the cluster that includes the primary CPU.

Function : bl1_plat_arch_setup() [mandatory]

```
Argument : void
Return   : void
```

This function performs any platform-specific and architectural setup that the platform requires. Platform-specific setup might include configuration of memory controllers and the interconnect.

In Arm standard platforms, this function enables the MMU.

This function helps fulfill requirement 2 above.

Function : bl1_platform_setup() [mandatory]

```
Argument : void
Return   : void
```

This function executes with the MMU and data caches enabled. It is responsible for performing any remaining platform-specific setup that can occur after the MMU and data cache have been enabled.

if support for multiple boot sources is required, it initializes the boot sequence used by plat_try_next_boot_source().

In Arm standard platforms, this function initializes the storage abstraction layer used to load the next bootloader image.

This function helps fulfill requirement 4 above.

Function : bl1_plat_sec_mem_layout() [mandatory]

```
Argument : void
Return   : meminfo *
```

This function should only be called on the cold boot path. It executes with the MMU and data caches enabled. The pointer returned by this function must point to a `meminfo` structure containing the extents and availability of secure RAM for the BL1 stage.

```
meminfo.total_base = Base address of secure RAM visible to BL1
meminfo.total_size = Size of secure RAM visible to BL1
```

This information is used by BL1 to load the BL2 image in secure RAM. BL1 also populates a similar structure to tell BL2 the extents of memory available for its own use.

This function helps fulfill requirements 4 and 5 above.

Function : bl1_plat_prepare_exit() [optional]

```
Argument : entry_point_info_t *
Return   : void
```

This function is called prior to exiting BL1 in response to the `BL1_SMC_RUN_IMAGE` SMC request raised by BL2. It should be used to perform platform specific clean up or bookkeeping operations before transferring control to the next image. It receives the address of the `entry_point_info_t` structure passed from BL2. This function runs with MMU disabled.

Function : bl1_plat_set_ep_info() [optional]

```
Argument : unsigned int image_id, entry_point_info_t *ep_info
Return   : void
```

This function allows platforms to override `ep_info` for the given `image_id`.

The default implementation just returns.

Function : bl1_plat_get_next_image_id() [optional]

```
Argument : void
Return   : unsigned int
```

This and the following function must be overridden to enable the FWU feature.

BL1 calls this function after platform setup to identify the next image to be loaded and executed. If the platform returns `BL2_IMAGE_ID` then BL1 proceeds with the normal boot sequence, which loads and executes BL2. If the platform returns a different image id, BL1 assumes that Firmware Update is required.

The default implementation always returns `BL2_IMAGE_ID`. The Arm development platforms override this function to detect if firmware update is required, and if so, return the first image in the firmware update process.

Function : bl1_plat_get_image_desc() [optional]

```
Argument : unsigned int image_id
Return   : image_desc_t *
```

BL1 calls this function to get the image descriptor information `image_desc_t` for the provided `image_id` from the platform.

The default implementation always returns a common BL2 image descriptor. Arm standard platforms return an image descriptor corresponding to BL2 or one of the firmware update images defined in the Trusted Board Boot Requirements specification.

Function : bl1_plat_handle_pre_image_load() [optional]

```
Argument : unsigned int image_id
Return   : int
```

This function can be used by the platforms to update/use image information corresponding to `image_id`. This function is invoked in BL1, both in cold boot and FWU code path, before loading the image.

Function : bl1_plat_handle_post_image_load() [optional]

```
Argument : unsigned int image_id
Return   : int
```

This function can be used by the platforms to update/use image information corresponding to `image_id`. This function is invoked in BL1, both in cold boot and FWU code path, after loading and authenticating the image.

The default weak implementation of this function calculates the amount of Trusted SRAM that can be used by BL2 and allocates a `meminfo_t` structure at the beginning of this free memory and populates it. The address of `meminfo_t` structure is updated in `arg1` of the entrypoint information to BL2.

Function : bl1_plat_fwu_done() [optional]

```
Argument : unsigned int image_id, uintptr_t image_src,
           unsigned int image_size
Return   : void
```

BL1 calls this function when the FWU process is complete. It must not return. The platform may override this function to take platform specific action, for example to initiate the normal boot flow.

The default implementation spins forever.

Function : bl1_plat_mem_check() [mandatory]

```
Argument : uintptr_t mem_base, unsigned int mem_size,
           unsigned int flags
Return   : int
```

BL1 calls this function while handling FWU related SMCs, more specifically when copying or authenticating an image. Its responsibility is to ensure that the region of memory identified by `mem_base` and `mem_size` is mapped in BL1, and that this memory corresponds to either a secure or non-secure memory region as indicated by the security state of the `flags` argument.

This function can safely assume that the value resulting from the addition of `mem_base` and `mem_size` fits into a `uintptr_t` type variable and does not overflow.

This function must return 0 on success, a non-null error code otherwise.

The default implementation of this function asserts therefore platforms must override it when using the FWU feature.

2.7.9 Boot Loader Stage 2 (BL2)

The BL2 stage is executed only by the primary CPU, which is determined in BL1 using the `platform_is_primary_cpu()` function. BL1 passed control to BL2 at `BL2_BASE`. BL2 executes in Secure EL1 and and invokes `plat_get_bl_image_load_info()` to retrieve the list of images to load from non-volatile storage to secure/non-secure RAM. After all the images are loaded then BL2 invokes `plat_get_next_bl_params()` to get the list of executable images to be passed to the next BL image.

The following functions must be implemented by the platform port to enable BL2 to perform the above tasks.

Function : bl2_early_platform_setup2() [mandatory]

```
Argument : u_register_t, u_register_t, u_register_t, u_register_t
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU. The 4 arguments are passed by BL1 to BL2 and these arguments are platform specific.

On Arm standard platforms, the arguments received are :

> arg0 - Points to load address of FW_CONFIG
>
> arg1 - `meminfo` structure populated by BL1. The platform copies the contents of `meminfo` as it may be subsequently overwritten by BL2.

On Arm standard platforms, this function also:

- Initializes a UART (PL011 console), which enables access to the `printf` family of functions in BL2.
- Initializes the storage abstraction layer used to load further bootloader images. It is necessary to do this early on platforms with a SCP_BL2 image, since the later `bl2_platform_setup` must be done after SCP_BL2 is loaded.

Function : bl2_plat_arch_setup() [mandatory]

```
Argument : void
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU.

The purpose of this function is to perform any architectural initialization that varies across platforms.

On Arm standard platforms, this function enables the MMU.

Function : bl2_platform_setup() [mandatory]

```
Argument : void
Return   : void
```

This function may execute with the MMU and data caches enabled if the platform port does the necessary initialization in `bl2_plat_arch_setup()`. It is only called by the primary CPU.

The purpose of this function is to perform any platform initialization specific to BL2.

In Arm standard platforms, this function performs security setup, including configuration of the TrustZone controller to allow non-secure masters access to most of DRAM. Part of DRAM is reserved for secure world use.

Function : bl2_plat_handle_pre_image_load() [optional]

```
Argument : unsigned int
Return   : int
```

This function can be used by the platforms to update/use image information for given `image_id`. This function is currently invoked in BL2 before loading each image.

Function : bl2_plat_handle_post_image_load() [optional]

```
Argument : unsigned int
Return   : int
```

This function can be used by the platforms to update/use image information for given `image_id`. This function is currently invoked in BL2 after loading each image.

Function : bl2_plat_preload_setup [optional]

```
Argument : void
Return   : void
```

This optional function performs any BL2 platform initialization required before image loading, that is not done later in bl2_platform_setup(). Specifically, if support for multiple boot sources is required, it initializes the boot sequence used by plat_try_next_boot_source().

Function : plat_try_next_boot_source() [optional]

```
Argument : void
Return   : int
```

This optional function passes to the next boot source in the redundancy sequence.

This function moves the current boot redundancy source to the next element in the boot sequence. If there are no more boot sources then it must return 0, otherwise it must return 1. The default implementation of this always returns 0.

2.7.10 Boot Loader Stage 2 (BL2) at EL3

When the platform has a non-TF-A Boot ROM it is desirable to jump directly to BL2 instead of TF-A BL1. In this case BL2 is expected to execute at EL3 instead of executing at EL1. Refer to the *Firmware Design* document for more information.

All mandatory functions of BL2 must be implemented, except the functions bl2_early_platform_setup and bl2_el3_plat_arch_setup, because their work is done now by bl2_el3_early_platform_setup and bl2_el3_plat_arch_setup. These functions should generally implement the bl1_plat_xxx() and bl2_plat_xxx() functionality combined.

Function : bl2_el3_early_platform_setup() [mandatory]

```
Argument : u_register_t, u_register_t, u_register_t, u_register_t
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU. This function receives four parameters which can be used by the platform to pass any needed information from the Boot ROM to BL2.

On Arm standard platforms, this function does the following:

- Initializes a UART (PL011 console), which enables access to the `printf` family of functions in BL2.
- Initializes the storage abstraction layer used to load further bootloader images. It is necessary to do this early on platforms with a SCP_BL2 image, since the later `bl2_platform_setup` must be done after SCP_BL2 is loaded.
- Initializes the private variables that define the memory layout used.

Function : bl2_el3_plat_arch_setup() [mandatory]

```
Argument : void
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU.

The purpose of this function is to perform any architectural initialization that varies across platforms.

On Arm standard platforms, this function enables the MMU.

Function : bl2_el3_plat_prepare_exit() [optional]

```
Argument : void
Return   : void
```

This function is called prior to exiting BL2 and run the next image. It should be used to perform platform specific clean up or bookkeeping operations before transferring control to the next image. This function runs with MMU disabled.

2.7.11 FWU Boot Loader Stage 2 (BL2U)

The AP Firmware Updater Configuration, BL2U, is an optional part of the FWU process and is executed only by the primary CPU. BL1 passes control to BL2U at `BL2U_BASE`. BL2U executes in Secure-EL1 and is responsible for:

1. (Optional) Transferring the optional SCP_BL2U binary image from AP secure memory to SCP RAM. BL2U uses the SCP_BL2U `image_info` passed by BL1. `SCP_BL2U_BASE` defines the address in AP secure memory where SCP_BL2U should be copied from. Subsequent handling of the SCP_BL2U image is implemented by the platform specific `bl2u_plat_handle_scp_bl2u()` function. If `SCP_BL2U_BASE` is not defined then this step is not performed.
2. Any platform specific setup required to perform the FWU process. For example, Arm standard platforms initialize the TZC controller so that the normal world can access DDR memory.

The following functions must be implemented by the platform port to enable BL2U to perform the tasks mentioned above.

Function : bl2u_early_platform_setup() [mandatory]

```
Argument : meminfo *mem_info, void *plat_info
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU. The arguments to this function is the address of the `meminfo` structure and platform specific info provided by BL1.

The platform may copy the contents of the `mem_info` and `plat_info` into private storage as the original memory may be subsequently overwritten by BL2U.

On Arm CSS platforms `plat_info` is interpreted as an `image_info_t` structure, to extract SCP_BL2U image information, which is then copied into a private variable.

Function : bl2u_plat_arch_setup() [mandatory]

```
Argument : void
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU.

The purpose of this function is to perform any architectural initialization that varies across platforms, for example enabling the MMU (since the memory map differs across platforms).

Function : bl2u_platform_setup() [mandatory]

```
Argument : void
Return   : void
```

This function may execute with the MMU and data caches enabled if the platform port does the necessary initialization in `bl2u_plat_arch_setup()`. It is only called by the primary CPU.

The purpose of this function is to perform any platform initialization specific to BL2U.

In Arm standard platforms, this function performs security setup, including configuration of the TrustZone controller to allow non-secure masters access to most of DRAM. Part of DRAM is reserved for secure world use.

Function : bl2u_plat_handle_scp_bl2u() [optional]

```
Argument : void
Return   : int
```

This function is used to perform any platform-specific actions required to handle the SCP firmware. Typically it transfers the image into SCP memory using a platform-specific protocol and waits until SCP executes it and signals to the Application Processor (AP) for BL2U execution to continue.

This function returns 0 on success, a negative error code otherwise. This function is included if SCP_BL2U_BASE is defined.

2.7.12 Boot Loader Stage 3-1 (BL31)

During cold boot, the BL31 stage is executed only by the primary CPU. This is determined in BL1 using the `platform_is_primary_cpu()` function. BL1 passes control to BL31 at `BL31_BASE`. During warm boot, BL31 is executed by all CPUs. BL31 executes at EL3 and is responsible for:

1. Re-initializing all architectural and platform state. Although BL1 performs some of this initialization, BL31 remains resident in EL3 and must ensure that EL3 architectural and platform state is completely initialized. It should make no assumptions about the system state when it receives control.
2. Passing control to a normal world BL image, pre-loaded at a platform- specific address by BL2. On ARM platforms, BL31 uses the `bl_params` list populated by BL2 in memory to do this.
3. Providing runtime firmware services. Currently, BL31 only implements a subset of the Power State Coordination Interface (PSCI) API as a runtime service. See Section 3.3 below for details of porting the PSCI implementation.
4. Optionally passing control to the BL32 image, pre-loaded at a platform- specific address by BL2. BL31 exports a set of APIs that allow runtime services to specify the security state in which the next image should be executed and run the corresponding image. On ARM platforms, BL31 uses the `bl_params` list populated by BL2 in memory to do this.

If BL31 is a reset vector, It also needs to handle the reset as specified in section 2.2 before the tasks described above.

The following functions must be implemented by the platform port to enable BL31 to perform the above tasks.

Function : bl31_early_platform_setup2() [mandatory]

```
Argument : u_register_t, u_register_t, u_register_t, u_register_t
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU. BL2 can pass 4 arguments to BL31 and these arguments are platform specific.

In Arm standard platforms, the arguments received are :

arg0 - The pointer to the head of *bl_params_t* list which is list of executable images following BL31,

arg1 - Points to load address of SOC_FW_CONFIG if present except in case of Arm FVP platform.

In case of Arm FVP platform, Points to load address of FW_CONFIG.

arg2 - Points to load address of HW_CONFIG if present

arg3 - A special value to verify platform parameters from BL2 to BL31. Not used in release builds.

The function runs through the *bl_param_t* list and extracts the entry point information for BL32 and BL33. It also performs the following:

- Initialize a UART (PL011 console), which enables access to the `printf` family of functions in BL31.
- Enable issuing of snoop and DVM (Distributed Virtual Memory) requests to the CCI slave interface corresponding to the cluster that includes the primary CPU.

Function : bl31_plat_arch_setup() [mandatory]

```
Argument : void
Return   : void
```

This function executes with the MMU and data caches disabled. It is only called by the primary CPU.

The purpose of this function is to perform any architectural initialization that varies across platforms.

On Arm standard platforms, this function enables the MMU.

Function : bl31_platform_setup() [mandatory]

```
Argument : void
Return   : void
```

This function may execute with the MMU and data caches enabled if the platform port does the necessary initialization in `bl31_plat_arch_setup()`. It is only called by the primary CPU.

The purpose of this function is to complete platform initialization so that both BL31 runtime services and normal world software can function correctly.

On Arm standard platforms, this function does the following:

- Initialize the generic interrupt controller.

  Depending on the GIC driver selected by the platform, the appropriate GICv2 or GICv3 initialization will be done, which mainly consists of:

  - Enable secure interrupts in the GIC CPU interface.
  - Disable the legacy interrupt bypass mechanism.
  - Configure the priority mask register to allow interrupts of all priorities to be signaled to the CPU interface.
  - Mark SGIs 8-15 and the other secure interrupts on the platform as secure.
  - Target all secure SPIs to CPU0.
  - Enable these secure interrupts in the GIC distributor.
  - Configure all other interrupts as non-secure.
  - Enable signaling of secure interrupts in the GIC distributor.

- Enable system-level implementation of the generic timer counter through the memory mapped interface.
- Grant access to the system counter timer module
- Initialize the power controller device.

  In particular, initialise the locks that prevent concurrent accesses to the power controller device.

Function : bl31_plat_runtime_setup() [optional]

```
Argument : void
Return   : void
```

The purpose of this function is allow the platform to perform any BL31 runtime setup just prior to BL31 exit during cold boot. The default weak implementation of this function will invoke `console_switch_state()` to switch console output to consoles marked for use in the `runtime` state.

Function : bl31_plat_get_next_image_ep_info() [mandatory]

```
Argument : uint32_t
Return   : entry_point_info *
```

This function may execute with the MMU and data caches enabled if the platform port does the necessary initializations in `bl31_plat_arch_setup()`.

This function is called by `bl31_main()` to retrieve information provided by BL2 for the next image in the security state specified by the argument. BL31 uses this information to pass control to that image in the specified security state. This function must return a pointer to the `entry_point_info` structure (that was copied during `bl31_early_platform_setup()`) if the image exists. It should return NULL otherwise.

Function : bl31_plat_enable_mmu [optional]

```
Argument : uint32_t
Return   : void
```

This function enables the MMU. The boot code calls this function with MMU and caches disabled. This function should program necessary registers to enable translation, and upon return, the MMU on the calling PE must be enabled.

The function must honor flags passed in the first argument. These flags are defined by the translation library, and can be found in the file `include/lib/xlat_tables/xlat_mmu_helpers.h`.

On DynamIQ systems, this function must not use stack while enabling MMU, which is how the function in xlat table library version 2 is implemented.

Function : plat_init_apkey [optional]

```
Argument : void
Return   : uint128_t
```

This function returns the 128-bit value which can be used to program ARMv8.3 pointer authentication keys.

The value should be obtained from a reliable source of randomness.

This function is only needed if ARMv8.3 pointer authentication is used in the Trusted Firmware by building with `BRANCH_PROTECTION` option set to non-zero.

Function : plat_get_syscnt_freq2() [mandatory]

```
Argument : void
Return   : unsigned int
```

This function is used by the architecture setup code to retrieve the counter frequency for the CPU's generic timer. This value will be programmed into the `CNTFRQ_EL0` register. In Arm standard platforms, it returns the base frequency of the system counter, which is retrieved from the first entry in the frequency modes table.

Function : plat_arm_set_twedel_scr_el3() [optional]

```
Argument : void
Return   : uint32_t
```

This function is used in v8.6+ systems to set the WFE trap delay value in SCR_EL3. If this function returns TWED_DISABLED or is left unimplemented, this feature is not enabled. The only hook provided is to set the TWED fields in SCR_EL3, there are similar fields in HCR_EL2, SCTLR_EL2, and SCTLR_EL1 to adjust the WFE trap delays in lower ELs and these fields should be set by the appropriate EL2 or EL1 code depending on the platform configuration.

#define : PLAT_PERCPU_BAKERY_LOCK_SIZE [optional]

When `USE_COHERENT_MEM = 0`, this constant defines the total memory (in bytes) aligned to the cache line boundary that should be allocated per-cpu to accommodate all the bakery locks.

If this constant is not defined when `USE_COHERENT_MEM = 0`, the linker calculates the size of the `bakery_lock` input section, aligns it to the nearest `CACHE_WRITEBACK_GRANULE`, multiplies it with `PLATFORM_CORE_COUNT` and stores the result in a linker symbol. This constant prevents a platform from relying on the linker and provide a more efficient mechanism for accessing per-cpu bakery lock information.

If this constant is defined and its value is not equal to the value calculated by the linker then a link time assertion is raised. A compile time assertion is raised if the value of the constant is not aligned to the cache line boundary.

SDEI porting requirements

The *SDEI* dispatcher requires the platform to provide the following macros and functions, of which some are optional, and some others mandatory.

Macros

Macro: PLAT_SDEI_NORMAL_PRI [mandatory]

This macro must be defined to the EL3 exception priority level associated with Normal *SDEI* events on the platform. This must have a higher value (therefore of lower priority) than `PLAT_SDEI_CRITICAL_PRI`.

Macro: PLAT_SDEI_CRITICAL_PRI [mandatory]

This macro must be defined to the EL3 exception priority level associated with Critical *SDEI* events on the platform. This must have a lower value (therefore of higher priority) than `PLAT_SDEI_NORMAL_PRI`.

Note: *SDEI* exception priorities must be the lowest among Secure priorities. Among the *SDEI* exceptions, Critical *SDEI* priority must be higher than Normal *SDEI* priority.

Functions

Function: int plat_sdei_validate_entry_point() [optional]

```
Argument: uintptr_t ep, unsigned int client_mode
Return: int
```

This function validates the entry point address of the event handler provided by the client for both event registration and *Complete and Resume SDEI* calls. The function ensures that the address is valid in the client translation regime.

The second argument is the exception level that the client is executing in. It can be Non-Secure EL1 or Non-Secure EL2.

The function must return `0` for successful validation, or `-1` upon failure.

The default implementation always returns `0`. On Arm platforms, this function translates the entry point address within the client translation regime and further ensures that the resulting physical address is located in Non-secure DRAM.

Function: void plat_sdei_handle_masked_trigger(uint64_t mpidr, unsigned int intr) [optional]

```
Argument: uint64_t
Argument: unsigned int
Return: void
```

SDEI specification requires that a PE comes out of reset with the events masked. The client therefore is expected to call `PE_UNMASK` to unmask *SDEI* events on the PE. No *SDEI* events can be dispatched until such time.

Should a PE receive an interrupt that was bound to an *SDEI* event while the events are masked on the PE, the dispatcher implementation invokes the function `plat_sdei_handle_masked_trigger`. The MPIDR of the PE that received the interrupt and the interrupt ID are passed as parameters.

The default implementation only prints out a warning message.

2.7.13 Power State Coordination Interface (in BL31)

The TF-A implementation of the PSCI API is based around the concept of a *power domain*. A *power domain* is a CPU or a logical group of CPUs which share some state on which power management operations can be performed as specified by PSCI. Each CPU in the system is assigned a cpu index which is a unique number between `0` and `PLATFORM_CORE_COUNT - 1`. The *power domains* are arranged in a hierarchical tree structure and each *power domain* can be identified in a system by the cpu index of any CPU that is part of that domain and a *power domain level*. A processing element (for example, a CPU) is at level 0. If the *power domain* node above a CPU is a logical grouping of CPUs that share some state, then level 1 is that group of CPUs (for example, a cluster), and level 2 is a group of clusters (for example, the system). More details on the power domain topology and its organization can be found in *PSCI Power Domain Tree Structure*.

BL31's platform initialization code exports a pointer to the platform-specific power management operations required for the PSCI implementation to function correctly. This information is populated in the `plat_psci_ops` structure. The PSCI implementation calls members of the `plat_psci_ops` structure for performing power management operations on the power domains. For example, the target CPU is specified by its `MPIDR` in a PSCI `CPU_ON` call. The `pwr_domain_on()` handler (if present) is called for the CPU power domain.

The `power-state` parameter of a PSCI `CPU_SUSPEND` call can be used to describe composite power states specific to a platform. The PSCI implementation defines a generic representation of the power-state parameter, which is an array of local power states where each index corresponds to a power domain level. Each entry contains the local power state the power domain at that power level could enter. It depends on the `validate_power_state()` handler to convert the power-state parameter (possibly encoding a composite power state) passed in a PSCI `CPU_SUSPEND` call to this representation.

The following functions form part of platform port of PSCI functionality.

Function : plat_psci_stat_accounting_start() [optional]

```
Argument : const psci_power_state_t *
Return   : void
```

This is an optional hook that platforms can implement for residency statistics accounting before entering a low power state. The `pwr_domain_state` field of `state_info` (first argument) can be inspected if stat accounting is done differently at CPU level versus higher levels. As an example, if the element at index 0 (CPU power level) in the `pwr_domain_state` array indicates a power down state, special hardware logic may be programmed in order to keep track of the residency statistics. For higher levels (array indices > 0), the residency statistics could be tracked in software using PMF. If `ENABLE_PMF` is set, the default implementation will use PMF to capture timestamps.

Function : plat_psci_stat_accounting_stop() [optional]

```
Argument : const psci_power_state_t *
Return   : void
```

This is an optional hook that platforms can implement for residency statistics accounting after exiting from a low power state. The `pwr_domain_state` field of `state_info` (first argument) can be inspected if stat accounting is done differently at CPU level versus higher levels. As an example, if the element at index 0 (CPU power level) in the `pwr_domain_state` array indicates a power down state, special hardware logic may be programmed in order to keep track of the residency statistics. For higher levels (array indices > 0), the residency statistics could be tracked in software using PMF. If `ENABLE_PMF` is set, the default implementation will use PMF to capture timestamps.

Function : plat_psci_stat_get_residency() [optional]

```
Argument : unsigned int, const psci_power_state_t *, unsigned int
Return   : u_register_t
```

This is an optional interface that is is invoked after resuming from a low power state and provides the time spent resident in that low power state by the power domain at a particular power domain level. When a CPU wakes up from suspend, all its parent power domain levels are also woken up. The generic PSCI code invokes this function for each parent power domain that is resumed and it identified by the `lvl` (first argument) parameter. The `state_info` (second argument) describes the low power state that the power domain has resumed from. The current CPU is the first CPU in the power domain to resume from the low power state and the `last_cpu_idx` (third parameter) is the index of the last CPU in the power domain to suspend and may be needed to calculate the residency for that power domain.

Function : plat_get_target_pwr_state() [optional]

```
Argument : unsigned int, const plat_local_state_t *, unsigned int
Return   : plat_local_state_t
```

The PSCI generic code uses this function to let the platform participate in state coordination during a power management operation. The function is passed a pointer to an array of platform specific local power state `states` (second argument) which contains the requested power state for each CPU at a particular power domain level `lvl` (first argument) within the power domain. The function is expected to traverse this array of upto `ncpus` (third argument) and return a coordinated target power state by the comparing all the requested power states. The target power state should not be deeper than any of the requested power states.

A weak definition of this API is provided by default wherein it assumes that the platform assigns a local state value in order of increasing depth of the power state i.e. for two power states X & Y, if X < Y then X represents a shallower power state than Y. As a result, the coordinated target local power state for a power domain will be the minimum of the requested local power state values.

Function : plat_get_power_domain_tree_desc() [mandatory]

```
Argument : void
Return   : const unsigned char *
```

This function returns a pointer to the byte array containing the power domain topology tree description. The format and method to construct this array are described in *PSCI Power Domain Tree Structure*. The BL31 PSCI initialization code requires this array to be described by the platform, either statically or dynamically, to initialize the power domain topology tree. In case the array is populated dynamically, then plat_core_pos_by_mpidr() and plat_my_core_pos() should also be implemented suitably so that the topology tree description matches the CPU indices returned by these APIs. These APIs together form the platform interface for the PSCI topology framework.

Function : plat_setup_psci_ops() [mandatory]

```
Argument : uintptr_t, const plat_psci_ops **
Return   : int
```

This function may execute with the MMU and data caches enabled if the platform port does the necessary initializations in `bl31_plat_arch_setup()`. It is only called by the primary CPU.

This function is called by PSCI initialization code. Its purpose is to let the platform layer know about the warm boot entrypoint through the `sec_entrypoint` (first argument) and to export handler routines for platform-specific psci power management actions by populating the passed pointer with a pointer to BL31's private `plat_psci_ops` structure.

A description of each member of this structure is given below. Please refer to the Arm FVP specific implementation of these handlers in `plat/arm/board/fvp/fvp_pm.c` as an example. For each PSCI function that the platform wants to support, the associated operation or operations in this structure must be provided and implemented (Refer section 4 of *Firmware Design* for the PSCI API supported in TF-A). To disable a PSCI function in a platform port, the operation should be removed from this structure instead of providing an empty implementation.

plat_psci_ops.cpu_standby()

Perform the platform-specific actions to enter the standby state for a cpu indicated by the passed argument. This provides a fast path for CPU standby wherein overheads of PSCI state management and lock acquisition is avoided. For this handler to be invoked by the PSCI `CPU_SUSPEND` API implementation, the suspend state type specified in the `power-state` parameter should be STANDBY and the target power domain level specified should be the CPU. The handler should put the CPU into a low power retention state (usually by issuing a wfi instruction) and ensure that it can be woken up from that state by a normal interrupt. The generic code expects the handler to succeed.

plat_psci_ops.pwr_domain_on()

Perform the platform specific actions to power on a CPU, specified by the `MPIDR` (first argument). The generic code expects the platform to return PSCI_E_SUCCESS on success or PSCI_E_INTERN_FAIL for any failure.

plat_psci_ops.pwr_domain_off()

Perform the platform specific actions to prepare to power off the calling CPU and its higher parent power domain levels as indicated by the `target_state` (first argument). It is called by the PSCI `CPU_OFF` API implementation.

The `target_state` encodes the platform coordinated target local power states for the CPU power domain and its parent power domain levels. The handler needs to perform power management operation corresponding to the local state at each power level.

For this handler, the local power state for the CPU power domain will be a power down state where as it could be either power down, retention or run state for the higher power domain levels depending on the result of state coordination. The generic code expects the handler to succeed.

plat_psci_ops.pwr_domain_suspend_pwrdown_early() [optional]

This optional function may be used as a performance optimization to replace or complement pwr_domain_suspend() on some platforms. Its calling semantics are identical to pwr_domain_suspend(), except the PSCI implementation only calls this function when suspending to a power down state, and it guarantees that data caches are enabled.

When HW_ASSISTED_COHERENCY = 0, the PSCI implementation disables data caches before calling pwr_domain_suspend(). If the target_state corresponds to a power down state and it is safe to perform some or all of the platform specific actions in that function with data caches enabled, it may be more efficient to move those actions to this function. When HW_ASSISTED_COHERENCY = 1, data caches remain enabled throughout, and so there is no advantage to moving platform specific actions to this function.

plat_psci_ops.pwr_domain_suspend()

Perform the platform specific actions to prepare to suspend the calling CPU and its higher parent power domain levels as indicated by the `target_state` (first argument). It is called by the PSCI `CPU_SUSPEND` API implementation.

The `target_state` has a similar meaning as described in the `pwr_domain_off()` operation. It encodes the platform coordinated target local power states for the CPU power domain and its parent power domain levels. The handler needs to perform power management operation corresponding to the local state at each power level. The generic code expects the handler to succeed.

The difference between turning a power domain off versus suspending it is that in the former case, the power domain is expected to re-initialize its state when it is next powered on (see `pwr_domain_on_finish()`). In the latter case, the power domain is expected to save enough state so that it can resume execution by restoring this state when its powered on (see `pwr_domain_suspend_finish()`).

When suspending a core, the platform can also choose to power off the GICv3 Redistributor and ITS through an implementation-defined sequence. To achieve this safely, the ITS context must be saved first. The architectural part is implemented by the `gicv3_its_save_disable()` helper, but most of the needed sequence is implementation defined and it is therefore the responsibility of the platform code to implement the necessary sequence. Then the GIC Redistributor context can be saved using the `gicv3_rdistif_save()` helper. Powering off the Redistributor requires the implementation to support it and it is the responsibility of the platform code to execute the right implementation defined sequence.

When a system suspend is requested, the platform can also make use of the `gicv3_distif_save()` helper to save the context of the GIC Distributor after it has saved the context of the Redistributors and ITS of all the cores in the system. The context of the Distributor can be large and may require it to be allocated in a special area if it cannot fit in the platform's global static data, for example in DRAM. The Distributor can then be powered down using an implementation-defined sequence.

plat_psci_ops.pwr_domain_pwr_down_wfi()

This is an optional function and, if implemented, is expected to perform platform specific actions including the `wfi` invocation which allows the CPU to powerdown. Since this function is invoked outside the PSCI locks, the actions performed in this hook must be local to the CPU or the platform must ensure that races between multiple CPUs cannot occur.

The `target_state` has a similar meaning as described in the `pwr_domain_off()` operation and it encodes the platform coordinated target local power states for the CPU power domain and its parent power domain levels. This function must not return back to the caller.

If this function is not implemented by the platform, PSCI generic implementation invokes `psci_power_down_wfi()` for power down.

plat_psci_ops.pwr_domain_on_finish()

This function is called by the PSCI implementation after the calling CPU is powered on and released from reset in response to an earlier PSCI `CPU_ON` call. It performs the platform-specific setup required to initialize enough state for this CPU to enter the normal world and also provide secure runtime firmware services.

The `target_state` (first argument) is the prior state of the power domains immediately before the CPU was turned on. It indicates which power domains above the CPU might require initialization due to having previously been in low power states. The generic code expects the handler to succeed.

plat_psci_ops.pwr_domain_on_finish_late() [optional]

This optional function is called by the PSCI implementation after the calling CPU is fully powered on with respective data caches enabled. The calling CPU and the associated cluster are guaranteed to be participating in coherency. This function gives the flexibility to perform any platform-specific actions safely, such as initialization or modification of shared data structures, without the overhead of explicit cache maintainace operations.

The `target_state` has a similar meaning as described in the `pwr_domain_on_finish()` operation. The generic code expects the handler to succeed.

plat_psci_ops.pwr_domain_suspend_finish()

This function is called by the PSCI implementation after the calling CPU is powered on and released from reset in response to an asynchronous wakeup event, for example a timer interrupt that was programmed by the CPU during the `CPU_SUSPEND` call or `SYSTEM_SUSPEND` call. It performs the platform-specific setup required to restore the saved state for this CPU to resume execution in the normal world and also provide secure runtime firmware services.

The `target_state` (first argument) has a similar meaning as described in the `pwr_domain_on_finish()` operation. The generic code expects the platform to succeed.

If the Distributor, Redistributors or ITS have been powered off as part of a suspend, their context must be restored in this function in the reverse order to how they were saved during suspend sequence.

plat_psci_ops.system_off()

This function is called by PSCI implementation in response to a `SYSTEM_OFF` call. It performs the platform-specific system poweroff sequence after notifying the Secure Payload Dispatcher.

plat_psci_ops.system_reset()

This function is called by PSCI implementation in response to a `SYSTEM_RESET` call. It performs the platform-specific system reset sequence after notifying the Secure Payload Dispatcher.

plat_psci_ops.validate_power_state()

This function is called by the PSCI implementation during the `CPU_SUSPEND` call to validate the `power_state` parameter of the PSCI API and if valid, populate it in `req_state` (second argument) array as power domain level specific local states. If the `power_state` is invalid, the platform must return PSCI_E_INVALID_PARAMS as error, which is propagated back to the normal world PSCI client.

plat_psci_ops.validate_ns_entrypoint()

This function is called by the PSCI implementation during the `CPU_SUSPEND`, `SYSTEM_SUSPEND` and `CPU_ON` calls to validate the non-secure `entry_point` parameter passed by the normal world. If the `entry_point` is invalid, the platform must return PSCI_E_INVALID_ADDRESS as error, which is propagated back to the normal world PSCI client.

plat_psci_ops.get_sys_suspend_power_state()

This function is called by the PSCI implementation during the `SYSTEM_SUSPEND` call to get the `req_state` parameter from platform which encodes the power domain level specific local states to suspend to system affinity level. The `req_state` will be utilized to do the PSCI state coordination and `pwr_domain_suspend()` will be invoked with the coordinated target state to enter system suspend.

plat_psci_ops.get_pwr_lvl_state_idx()

This is an optional function and, if implemented, is invoked by the PSCI implementation to convert the `local_state` (first argument) at a specified `pwr_lvl` (second argument) to an index between 0 and `PLAT_MAX_PWR_LVL_STATES` - 1. This function is only needed if the platform supports more than two local power states at each power domain level, that is `PLAT_MAX_PWR_LVL_STATES` is greater than 2, and needs to account for these local power states.

plat_psci_ops.translate_power_state_by_mpidr()

This is an optional function and, if implemented, verifies the `power_state` (second argument) parameter of the PSCI API corresponding to a target power domain. The target power domain is identified by using both `MPIDR` (first argument) and the power domain level encoded in `power_state`. The power domain level specific local states are to be extracted from `power_state` and be populated in the `output_state` (third argument) array. The functionality is similar to the `validate_power_state` function described above and is envisaged to be used in case the validity of `power_state` depend on the targeted power domain. If the `power_state` is invalid for the targeted power domain, the platform must return PSCI_E_INVALID_PARAMS as error. If this function is not implemented, then the generic implementation relies on `validate_power_state` function to translate the `power_state`.

This function can also be used in case the platform wants to support local power state encoding for `power_state` parameter of PSCI_STAT_COUNT/RESIDENCY APIs as described in Section 5.18 of PSCI.

plat_psci_ops.get_node_hw_state()

This is an optional function. If implemented this function is intended to return the power state of a node (identified by the first parameter, the `MPIDR`) in the power domain topology (identified by the second parameter, `power_level`), as retrieved from a power controller or equivalent component on the platform. Upon successful completion, the implementation must map and return the final status among `HW_ON`, `HW_OFF` or `HW_STANDBY`. Upon encountering failures, it must return either `PSCI_E_INVALID_PARAMS` or `PSCI_E_NOT_SUPPORTED` as appropriate.

Implementations are not expected to handle `power_levels` greater than `PLAT_MAX_PWR_LVL`.

plat_psci_ops.system_reset2()

This is an optional function. If implemented this function is called during the `SYSTEM_RESET2` call to perform a reset based on the first parameter `reset_type` as specified in PSCI. The parameter `cookie` can be used to pass additional reset information. If the `reset_type` is not supported, the function must return `PSCI_E_NOT_SUPPORTED`. For architectural resets, all failures must return `PSCI_E_INVALID_PARAMETERS` and vendor reset can return other PSCI error codes as defined in PSCI. On success this function will not return.

plat_psci_ops.write_mem_protect()

This is an optional function. If implemented it enables or disables the `MEM_PROTECT` functionality based on the value of `val`. A non-zero value enables `MEM_PROTECT` and a value of zero disables it. Upon encountering failures it must return a negative value and on success it must return 0.

plat_psci_ops.read_mem_protect()

This is an optional function. If implemented it returns the current state of `MEM_PROTECT` via the `val` parameter. Upon encountering failures it must return a negative value and on success it must return 0.

plat_psci_ops.mem_protect_chk()

This is an optional function. If implemented it checks if a memory region defined by a base address `base` and with a size of `length` bytes is protected by `MEM_PROTECT`. If the region is protected then it must return 0, otherwise it must return a negative number.

2.7.14 Interrupt Management framework (in BL31)

BL31 implements an Interrupt Management Framework (IMF) to manage interrupts generated in either security state and targeted to EL1 or EL2 in the non-secure state or EL3/S-EL1 in the secure state. The design of this framework is described in the *Interrupt Management Framework*

A platform should export the following APIs to support the IMF. The following text briefly describes each API and its implementation in Arm standard platforms. The API implementation depends upon the type of interrupt controller present in the platform. Arm standard platform layer supports both Arm Generic Interrupt Controller version 2.0 (GICv2) and 3.0 (GICv3). Juno builds the Arm platform layer to use GICv2 and the FVP can be configured to use either GICv2 or GICv3 depending on the build flag `FVP_USE_GIC_DRIVER` (See *Arm FVP Platform Specific Build Options* for more details).

See also: *Interrupt Controller Abstraction APIs*.

Function : plat_interrupt_type_to_line() [mandatory]

```
Argument : uint32_t, uint32_t
Return   : uint32_t
```

The Arm processor signals an interrupt exception either through the IRQ or FIQ interrupt line. The specific line that is signaled depends on how the interrupt controller (IC) reports different interrupt types from an execution context in either security state. The IMF uses this API to determine which interrupt line the platform IC uses to signal each type of interrupt supported by the framework from a given security state. This API must be invoked at EL3.

The first parameter will be one of the `INTR_TYPE_*` values (see *Interrupt Management Framework*) indicating the target type of the interrupt, the second parameter is the security state of the originating execution context. The return result is the bit position in the `SCR_EL3` register of the respective interrupt trap: IRQ=1, FIQ=2.

In the case of Arm standard platforms using GICv2, S-EL1 interrupts are configured as FIQs and Non-secure interrupts as IRQs from either security state.

In the case of Arm standard platforms using GICv3, the interrupt line to be configured depends on the security state of the execution context when the interrupt is signalled and are as follows:

- The S-EL1 interrupts are signaled as IRQ in S-EL0/1 context and as FIQ in NS-EL0/1/2 context.
- The Non secure interrupts are signaled as FIQ in S-EL0/1 context and as IRQ in the NS-EL0/1/2 context.
- The EL3 interrupts are signaled as FIQ in both S-EL0/1 and NS-EL0/1/2 context.

Function : plat_ic_get_pending_interrupt_type() [mandatory]

```
Argument : void
Return   : uint32_t
```

This API returns the type of the highest priority pending interrupt at the platform IC. The IMF uses the interrupt type to retrieve the corresponding handler function. `INTR_TYPE_INVAL` is returned when there is no interrupt pending. The valid interrupt types that can be returned are `INTR_TYPE_EL3`, `INTR_TYPE_S_EL1` and `INTR_TYPE_NS`. This API must be invoked at EL3.

In the case of Arm standard platforms using GICv2, the *Highest Priority Pending Interrupt Register* (`GICC_HPPIR`) is read to determine the id of the pending interrupt. The type of interrupt depends upon the id value as follows.

1. id < 1022 is reported as a S-EL1 interrupt
2. id = 1022 is reported as a Non-secure interrupt.
3. id = 1023 is reported as an invalid interrupt type.

In the case of Arm standard platforms using GICv3, the system register `ICC_HPPIR0_EL1`, *Highest Priority Pending group 0 Interrupt Register*, is read to determine the id of the pending interrupt. The type of interrupt depends upon the id value as follows.

1. id = `PENDING_G1S_INTID` (1020) is reported as a S-EL1 interrupt
2. id = `PENDING_G1NS_INTID` (1021) is reported as a Non-secure interrupt.
3. id = `GIC_SPURIOUS_INTERRUPT` (1023) is reported as an invalid interrupt type.
4. All other interrupt id's are reported as EL3 interrupt.

Function : plat_ic_get_pending_interrupt_id() [mandatory]

```
Argument : void
Return   : uint32_t
```

This API returns the id of the highest priority pending interrupt at the platform IC. `INTR_ID_UNAVAILABLE` is returned when there is no interrupt pending.

In the case of Arm standard platforms using GICv2, the *Highest Priority Pending Interrupt Register* (`GICC_HPPIR`) is read to determine the id of the pending interrupt. The id that is returned by API depends upon the value of the id read from the interrupt controller as follows.

1. id < 1022. id is returned as is.
2. id = 1022. The *Aliased Highest Priority Pending Interrupt Register* (`GICC_AHPPIR`) is read to determine the id of the non-secure interrupt. This id is returned by the API.
3. id = 1023. `INTR_ID_UNAVAILABLE` is returned.

In the case of Arm standard platforms using GICv3, if the API is invoked from EL3, the system register `ICC_HPPIR0_EL1`, *Highest Priority Pending Interrupt group 0 Register*, is read to determine the id of the pending interrupt. The id that is returned by API depends upon the value of the id read from the interrupt controller as follows.

1. id < `PENDING_G1S_INTID` (1020). id is returned as is.
2. id = `PENDING_G1S_INTID` (1020) or `PENDING_G1NS_INTID` (1021). The system register `ICC_HPPIR1_EL1`, *Highest Priority Pending Interrupt group 1 Register* is read to determine the id of the group 1 interrupt. This id is returned by the API as long as it is a valid interrupt id

3. If the id is any of the special interrupt identifiers, `INTR_ID_UNAVAILABLE` is returned.

When the API invoked from S-EL1 for GICv3 systems, the id read from system register `ICC_HPPIR1_EL1`, *Highest Priority Pending group 1 Interrupt Register*, is returned if is not equal to GIC_SPURIOUS_INTERRUPT (1023) else `INTR_ID_UNAVAILABLE` is returned.

Function : plat_ic_acknowledge_interrupt() [mandatory]

```
Argument : void
Return   : uint32_t
```

This API is used by the CPU to indicate to the platform IC that processing of the highest pending interrupt has begun. It should return the raw, unmodified value obtained from the interrupt controller when acknowledging an interrupt. The actual interrupt number shall be extracted from this raw value using the API *plat_ic_get_interrupt_id()<plat_ic_get_interrupt_id>*.

This function in Arm standard platforms using GICv2, reads the *Interrupt Acknowledge Register* (`GICC_IAR`). This changes the state of the highest priority pending interrupt from pending to active in the interrupt controller. It returns the value read from the `GICC_IAR`, unmodified.

In the case of Arm standard platforms using GICv3, if the API is invoked from EL3, the function reads the system register `ICC_IAR0_EL1`, *Interrupt Acknowledge Register group 0*. If the API is invoked from S-EL1, the function reads the system register `ICC_IAR1_EL1`, *Interrupt Acknowledge Register group 1*. The read changes the state of the highest pending interrupt from pending to active in the interrupt controller. The value read is returned unmodified.

The TSP uses this API to start processing of the secure physical timer interrupt.

Function : plat_ic_end_of_interrupt() [mandatory]

```
Argument : uint32_t
Return   : void
```

This API is used by the CPU to indicate to the platform IC that processing of the interrupt corresponding to the id (passed as the parameter) has finished. The id should be the same as the id returned by the `plat_ic_acknowledge_interrupt()` API.

Arm standard platforms write the id to the *End of Interrupt Register* (`GICC_EOIR`) in case of GICv2, and to `ICC_EOIR0_EL1` or `ICC_EOIR1_EL1` system register in case of GICv3 depending on where the API is invoked from, EL3 or S-EL1. This deactivates the corresponding interrupt in the interrupt controller.

The TSP uses this API to finish processing of the secure physical timer interrupt.

Function : plat_ic_get_interrupt_type() [mandatory]

```
Argument : uint32_t
Return   : uint32_t
```

This API returns the type of the interrupt id passed as the parameter. `INTR_TYPE_INVAL` is returned if the id is invalid. If the id is valid, a valid interrupt type (one of `INTR_TYPE_EL3`, `INTR_TYPE_S_EL1` and `INTR_TYPE_NS`) is returned depending upon how the interrupt has been configured by the platform IC. This API must be invoked at EL3.

Arm standard platforms using GICv2 configures S-EL1 interrupts as Group0 interrupts and Non-secure interrupts as Group1 interrupts. It reads the group value corresponding to the interrupt id from the relevant *Interrupt Group Register* (`GICD_IGROUPRn`). It uses the group value to determine the type of interrupt.

In the case of Arm standard platforms using GICv3, both the *Interrupt Group Register* (`GICD_IGROUPRn`) and *Interrupt Group Modifier Register* (`GICD_IGRPMODRn`) is read to figure out whether the interrupt is configured as Group 0 secure interrupt, Group 1 secure interrupt or Group 1 NS interrupt.

2.7.15 Crash Reporting mechanism (in BL31)

BL31 implements a crash reporting mechanism which prints the various registers of the CPU to enable quick crash analysis and debugging. This mechanism relies on the platform implementing `plat_crash_console_init`, `plat_crash_console_putc` and `plat_crash_console_flush`.

The file `plat/common/aarch64/crash_console_helpers.S` contains sample implementation of all of them. Platforms may include this file to their makefiles in order to benefit from them. By default, they will cause the crash output to be routed over the normal console infrastructure and get printed on consoles configured to output in crash state. `console_set_scope()` can be used to control whether a console is used for crash output.

Note: Platforms are responsible for making sure that they only mark consoles for use in the crash scope that are able to support this, i.e. that are written in assembly and conform with the register clobber rules for putc() (x0-x2, x16-x17) and flush() (x0-x3, x16-x17) crash callbacks.

In some cases (such as debugging very early crashes that happen before the normal boot console can be set up), platforms may want to control crash output more explicitly. These platforms may instead provide custom implementations for these. They are executed outside of a C environment and without a stack. Many console drivers provide functions named `console_xxx_core_init/putc/flush` that are designed to be used by these functions. See Arm platforms (like juno) for an example of this.

Function : plat_crash_console_init [mandatory]

```
Argument : void
Return   : int
```

This API is used by the crash reporting mechanism to initialize the crash console. It must only use the general purpose registers x0 through x7 to do the initialization and returns 1 on success.

Function : plat_crash_console_putc [mandatory]

```
Argument : int
Return   : int
```

This API is used by the crash reporting mechanism to print a character on the designated crash console. It must only use general purpose registers x1 and x2 to do its work. The parameter and the return value are in general purpose register x0.

Function : plat_crash_console_flush [mandatory]

```
Argument : void
Return   : void
```

This API is used by the crash reporting mechanism to force write of all buffered data on the designated crash console. It should only use general purpose registers x0 through x5 to do its work.

2.7.16 External Abort handling and RAS Support

Function : plat_ea_handler

```
Argument : int
Argument : uint64_t
Argument : void *
Argument : void *
Argument : uint64_t
Return   : void
```

This function is invoked by the RAS framework for the platform to handle an External Abort received at EL3. The intention of the function is to attempt to resolve the cause of External Abort and return; if that's not possible, to initiate orderly shutdown of the system.

The first parameter (`int ea_reason`) indicates the reason for External Abort. Its value is one of `ERROR_EA_*` constants defined in `ea_handle.h`.

The second parameter (`uint64_t syndrome`) is the respective syndrome presented to EL3 after having received the External Abort. Depending on the nature of the abort (as can be inferred from the `ea_reason` parameter), this can be the content of either `ESR_EL3` or `DISR_EL1`.

The third parameter (`void *cookie`) is unused for now. The fourth parameter (`void *handle`) is a pointer to the preempted context. The fifth parameter (`uint64_t flags`) indicates the preempted security state. These parameters are received from the top-level exception handler.

If `RAS_EXTENSION` is set to `1`, the default implementation of this function iterates through RAS handlers registered by the platform. If any of the RAS handlers resolve the External Abort, no further action is taken.

If `RAS_EXTENSION` is set to `0`, or if none of the platform RAS handlers could resolve the External Abort, the default implementation prints an error message, and panics.

Function : plat_handle_uncontainable_ea

```
Argument : int
Argument : uint64_t
Return   : void
```

This function is invoked by the RAS framework when an External Abort of Uncontainable type is received at EL3. Due to the critical nature of Uncontainable errors, the intention of this function is to initiate orderly shutdown of the system, and is not expected to return.

This function must be implemented in assembly.

The first and second parameters are the same as that of `plat_ea_handler`.

The default implementation of this function calls `report_unhandled_exception`.

Function : plat_handle_double_fault

```
Argument : int
Argument : uint64_t
Return   : void
```

This function is invoked by the RAS framework when another External Abort is received at EL3 while one is already being handled. I.e., a call to `plat_ea_handler` is outstanding. Due to its critical nature, the intention of this function is to initiate orderly shutdown of the system, and is not expected recover or return.

This function must be implemented in assembly.

The first and second parameters are the same as that of `plat_ea_handler`.

The default implementation of this function calls `report_unhandled_exception`.

Function : plat_handle_el3_ea

```
Return   : void
```

This function is invoked when an External Abort is received while executing in EL3. Due to its critical nature, the intention of this function is to initiate orderly shutdown of the system, and is not expected recover or return.

This function must be implemented in assembly.

The default implementation of this function calls `report_unhandled_exception`.

2.7.17 Build flags

There are some build flags which can be defined by the platform to control inclusion or exclusion of certain BL stages from the FIP image. These flags need to be defined in the platform makefile which will get included by the build system.

- **NEED_BL33** By default, this flag is defined `yes` by the build system and `BL33` build option should be supplied as a build option. The platform has the option of excluding the BL33 image in the `fip` image by defining this flag to `no`. If any of the options `EL3_PAYLOAD_BASE` or `PRELOADED_BL33_BASE` are used, this flag will be set to `no` automatically.

2.7.18 Platform include paths

Platforms are allowed to add more include paths to be passed to the compiler. The `PLAT_INCLUDES` variable is used for this purpose. This is needed in particular for the file `platform_def.h`.

Example:

```
PLAT_INCLUDES  += -Iinclude/plat/myplat/include
```

2.7.19 C Library

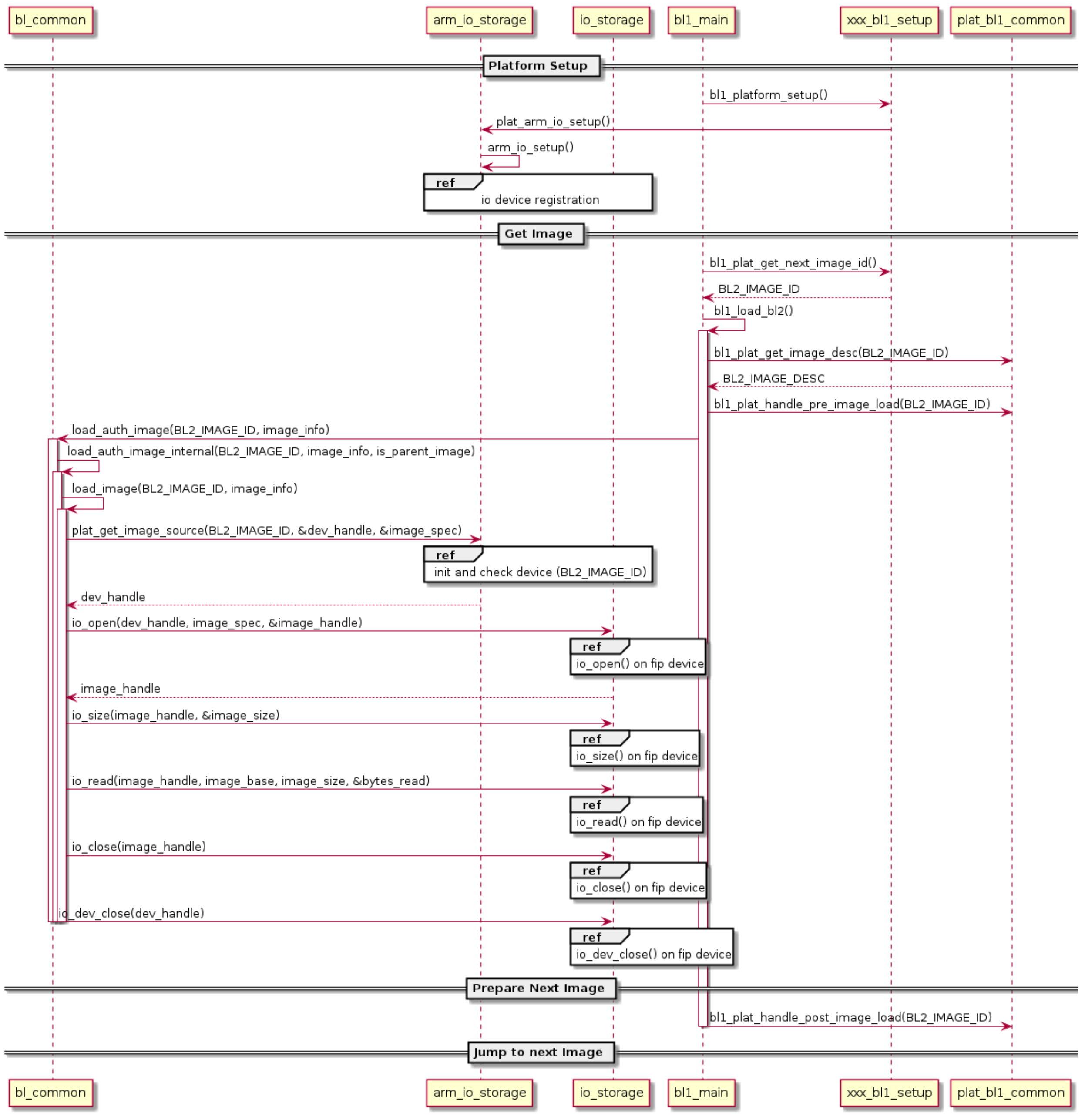
To avoid subtle toolchain behavioral dependencies, the header files provided by the compiler are not used. The software is built with the `-nostdinc` flag to ensure no headers are included from the toolchain inadvertently. Instead the required headers are included in the TF-A source tree. The library only contains those C library definitions required by the local implementation. If more functionality is required, the needed library functions will need to be added to the local implementation.

Some C headers have been obtained from FreeBSD and SCC, while others have been written specifically for TF-A. Some implementation files have been obtained from FreeBSD, others have been written specifically for TF-A as well. The files can be found in `include/lib/libc` and `lib/libc`.

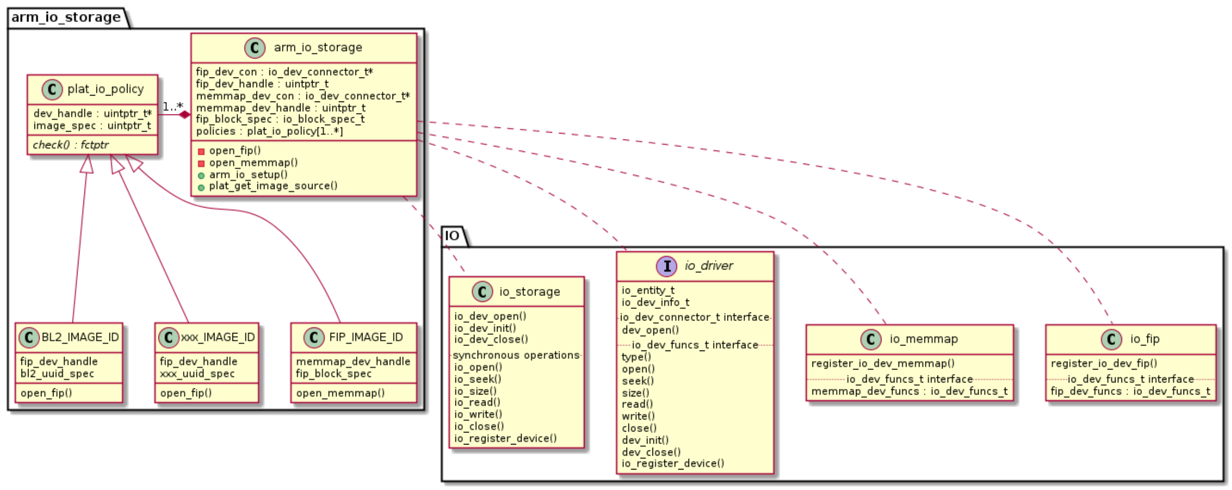
SCC can be found in http://www.simple-cc.org/. A copy of the FreeBSD sources can be obtained from http://github.com/freebsd/freebsd.

2.7.20 Storage abstraction layer

In order to improve platform independence and portability a storage abstraction layer is used to load data from non-volatile platform storage. Currently storage access is only required by BL1 and BL2 phases and performed inside the `load_image()` function in `bl_common.c`.

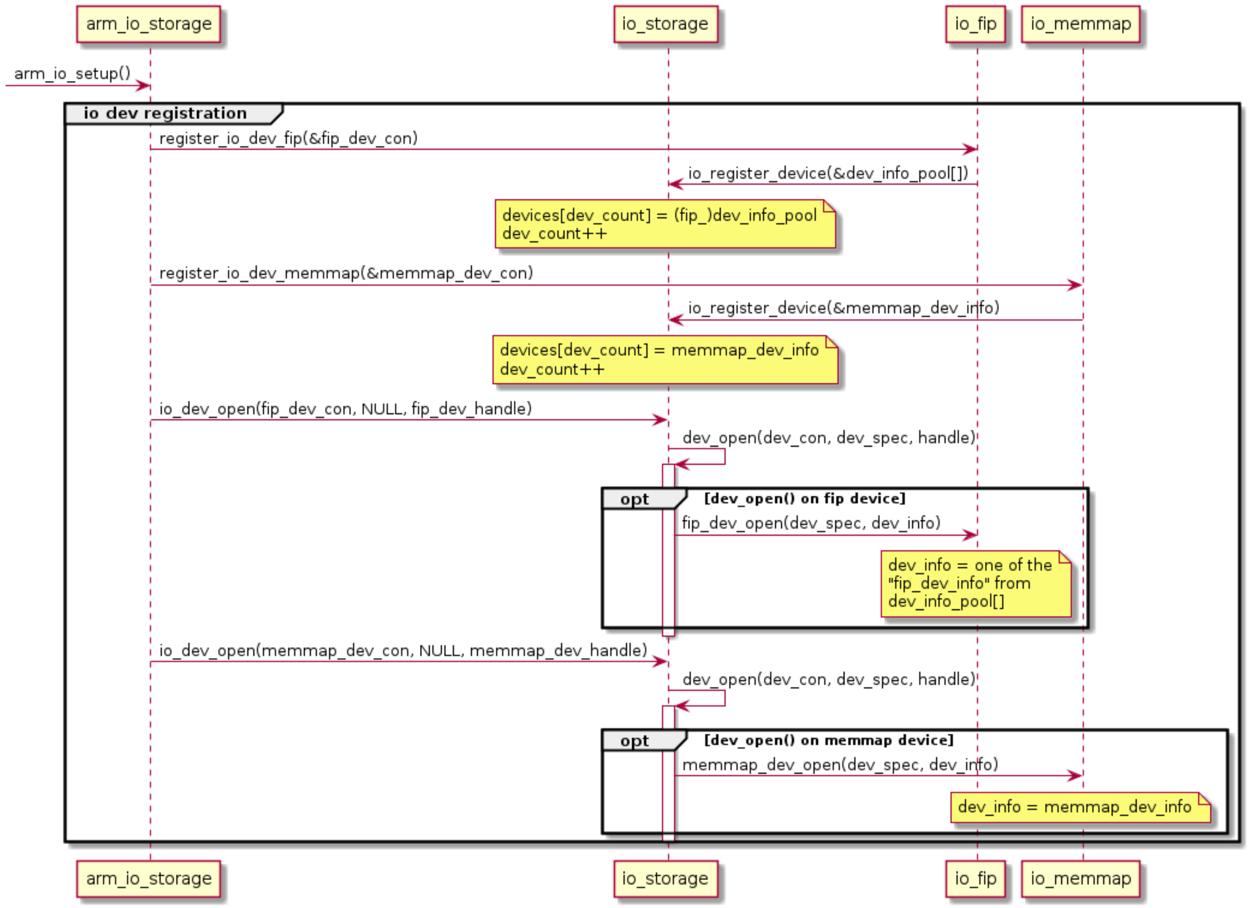

bl_common
arm_io_storage
io_storage
bl1_main
xxx_bl1_setup
plat_bl1_common
Platform Setup
bl1_platform_setup()
plat_arm_io_setup()
arm_io_setup()
ref
io device registration
Get Image
bl1_plat_get_next_image_id()
BL2_IMAGE_ID
bl1_load_bl2()
bl1_plat_get_image_desc(BL2_IMAGE_ID)
BL2_IMAGE_DESC
bl1_plat_handle_pre_image_load(BL2_IMAGE_ID)
load_auth_image(BL2_IMAGE_ID, image_info)
load_auth_image_internal(BL2_IMAGE_ID, image_info, is_parent_image)
load_image(BL2_IMAGE_ID, image_info)
plat_get_image_source(BL2_IMAGE_ID, &dev_handle, &image_spec)
ref
init and check device (BL2_IMAGE_ID)
dev_handle
io_open(dev_handle, image_spec, &image_handle)
ref
io_open() on fip device
image_handle
io_size(image_handle, &image_size)
ref
io_size() on fip device
io_read(image_handle, image_base, image_size, &bytes_read)
ref
io_read() on fip device
io_close(image_handle)
ref
io_close() on fip device
io_dev_close(dev_handle)
ref
io_dev_close() on fip device
Prepare Next Image
bl1_plat_handle_post_image_load(BL2_IMAGE_ID)
Jump to next Image

It is mandatory to implement at least one storage driver. For the Arm development platforms the Firmware Image Package (FIP) driver is provided as the default means to load data from storage (see *Firmware Image Package (FIP)*). The storage layer is described in the header file `include/drivers/io/io_storage.h`. The implementation of the common library is in `drivers/io/io_storage.c` and the driver files are located in `drivers/io/`.

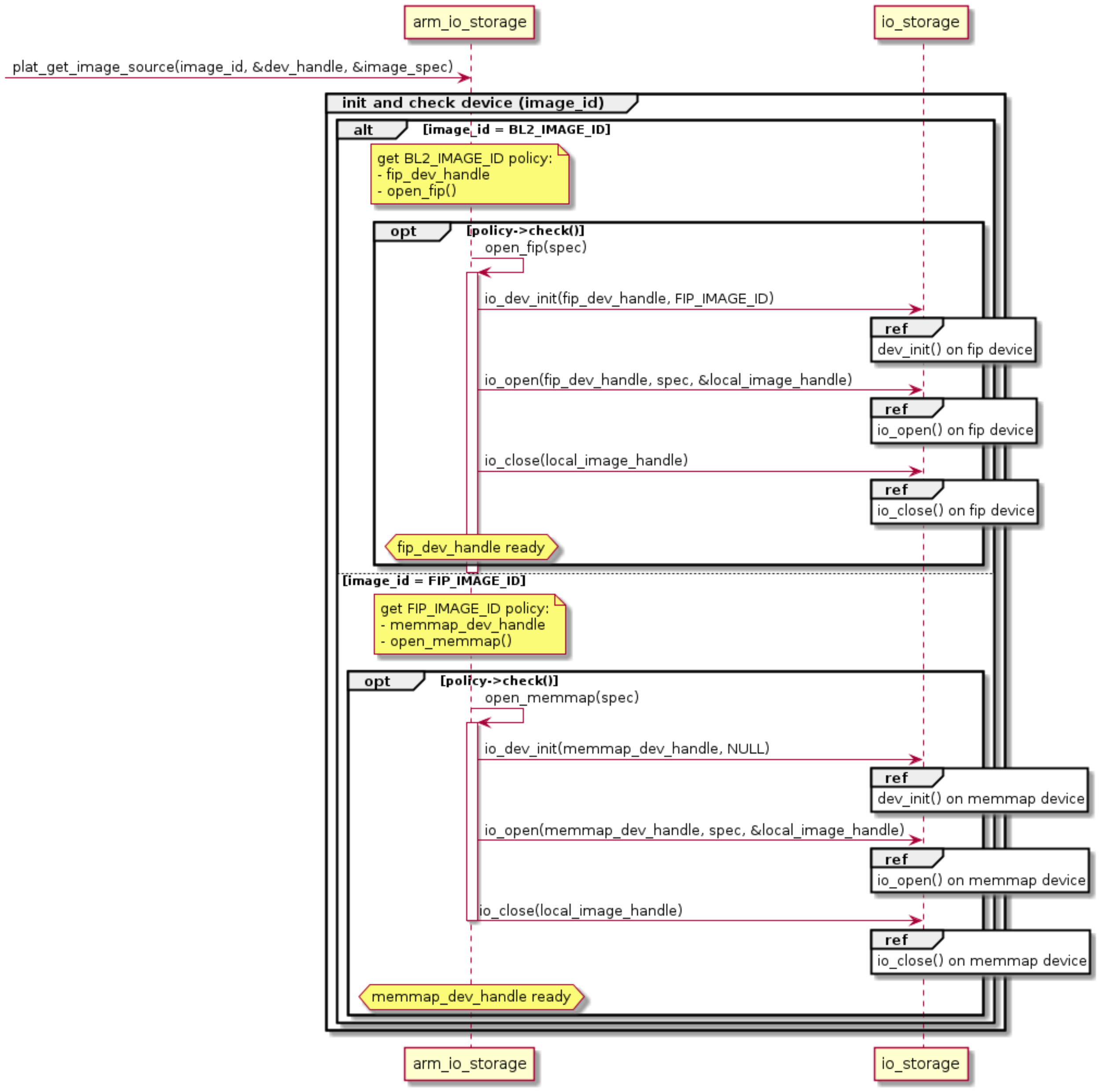


Each IO driver must provide `io_dev_*` structures, as described in `drivers/io/io_driver.h`. These are returned via a mandatory registration function that is called on platform initialization. The semi-hosting driver implementation in `io_semihosting.c` can be used as an example.

Each platform should register devices and their drivers via the storage abstraction layer. These drivers then need to be initialized by bootloader phases as required in their respective `blx_platform_setup()` functions.


arm_io_storage
io_storage
io_fip
io_memmap
arm_io_setup()
io dev registration
register_io_dev_fip(&fip_dev_con)
io_register_device(&dev_info_pool[])
devices[dev_count] = (fip_)dev_info_pool
dev_count++
register_io_dev_memmap(&memmap_dev_con)
io_register_device(&memmap_dev_info)
devices[dev_count] = memmap_dev_info
dev_count++
io_dev_open(fip_dev_con, NULL, fip_dev_handle)
dev_open(dev_con, dev_spec, handle)
opt
[dev_open() on fip device]
fip_dev_open(dev_spec, dev_info)
dev_info = one of the
"fip_dev_info" from
dev_info_pool[]
io_dev_open(memmap_dev_con, NULL, memmap_dev_handle)
dev_open(dev_con, dev_spec, handle)
opt
[dev_open() on memmap device]
memmap_dev_open(dev_spec, dev_info)
dev_info = memmap_dev_info
arm_io_storage
io_storage
io_fip
io_memmap

The storage abstraction layer provides mechanisms (`io_dev_init()`) to initialize storage devices before IO operations are called.


arm_io_storage
io_storage
plat_get_image_source(image_id, &dev_handle, &image_spec)
init and check device (image_id)
alt [image_id = BL2_IMAGE_ID]
get BL2_IMAGE_ID policy:
- fip_dev_handle
- open_fip()
opt [policy->check()]
open_fip(spec)
io_dev_init(fip_dev_handle, FIP_IMAGE_ID)
ref dev_init() on fip device
io_open(fip_dev_handle, spec, &local_image_handle)
ref io_open() on fip device
io_close(local_image_handle)
ref io_close() on fip device
fip_dev_handle ready
[image_id = FIP_IMAGE_ID]
get FIP_IMAGE_ID policy:
- memmap_dev_handle
- open_memmap()
opt [policy->check()]
open_memmap(spec)
io_dev_init(memmap_dev_handle, NULL)
ref dev_init() on memmap device
io_open(memmap_dev_handle, spec, &local_image_handle)
ref io_open() on memmap device
io_close(local_image_handle)
ref io_close() on memmap device
memmap_dev_handle ready
arm_io_storage
io_storage

The basic operations supported by the layer include `open()`, `close()`, `read()`, `write()`, `size()` and `seek()`. Drivers do not have to implement all operations, but each platform must provide at least one driver for a device capable of supporting generic operations such as loading a bootloader image.

The current implementation only allows for known images to be loaded by the firmware. These images are specified by using their identifiers, as defined in `include/plat/common/common_def.h` (or a separate header file included from there). The platform layer (`plat_get_image_source()`) then returns a reference to a device and a driver-specific `spec` which will be understood by the driver to allow access to the image data.

The layer is designed in such a way that is it possible to chain drivers with other drivers. For example, file-system drivers may be implemented on top of physical block devices, both represented by IO devices with corresponding drivers. In such a case, the file-system "binding" with the block device may be deferred until the file-system device is initialised.

The abstraction currently depends on structures being statically allocated by the drivers and callers, as the system does not yet provide a means of dynamically allocating memory. This may also have the affect of limiting the amount of open resources per driver.

2.8 PSCI Library Integration guide for Armv8-A AArch32 systems

This document describes the PSCI library interface with a focus on how to integrate with a suitable Trusted OS for an Armv8-A AArch32 system. The PSCI Library implements the PSCI Standard as described in PSCI spec and is meant to be integrated with EL3 Runtime Software which invokes the PSCI Library interface appropriately. **EL3 Runtime Software** refers to software executing at the highest secure privileged mode, which is EL3 in AArch64 or Secure SVC/ Monitor mode in AArch32, and provides runtime services to the non-secure world. The runtime service request is made via SMC (Secure Monitor Call) and the call must adhere to SMCCC. In AArch32, EL3 Runtime Software may additionally include Trusted OS functionality. A minimal AArch32 Secure Payload, SP-MIN, is provided in Trusted Firmware-A (TF-A) to illustrate the usage and integration of the PSCI library. The description of PSCI library interface and its integration with EL3 Runtime Software in this document is targeted towards AArch32 systems.

2.8.1 Generic call sequence for PSCI Library interface (AArch32)

The generic call sequence of PSCI Library interfaces (see *PSCI Library Interface*) during cold boot in AArch32 system is described below:

1. After cold reset, the EL3 Runtime Software performs its cold boot initialization including the PSCI library pre-requisites mentioned in *PSCI Library Interface*, and also the necessary platform setup.
2. Call `psci_setup()` in Monitor mode.
3. Optionally call `psci_register_spd_pm_hook()` to register callbacks to do bookkeeping for the EL3 Runtime Software during power management.
4. Call `psci_prepare_next_non_secure_ctx()` to initialize the non-secure CPU context.
5. Get the non-secure `cpu_context_t` for the current CPU by calling `cm_get_context()` , then programming the registers in the non-secure context and exiting to non-secure world. If the EL3 Runtime Software needs additional configuration to be set for non-secure context, like routing FIQs to the secure world, the values of the registers can be modified prior to programming. See *PSCI CPU context management* for more details on CPU context management.

The generic call sequence of PSCI library interfaces during warm boot in AArch32 systems is described below:

1. After warm reset, the EL3 Runtime Software performs the necessary warm boot initialization including the PSCI library pre-requisites mentioned in *PSCI Library Interface* (Note that the Data cache **must not** be enabled).
2. Call `psci_warmboot_entrypoint()` in Monitor mode. This interface initializes/restores the non-secure CPU context as well.
3. Do step 5 of the cold boot call sequence described above.

The generic call sequence of PSCI library interfaces on receipt of a PSCI SMC on an AArch32 system is described below:

1. On receipt of an SMC, save the register context as per SMCCC.
2. If the SMC function identifier corresponds to a SMC32 PSCI API, construct the appropriate arguments and call the `psci_smc_handler()` interface. The invocation may or may not return back to the caller depending on whether the PSCI API resulted in power down of the CPU.

3. If psci_smc_handler() returns, populate the return value in R0 (AArch32)/ X0 (AArch64) and restore other registers as per SMCCC.

2.8.2 PSCI CPU context management

PSCI library is in charge of initializing/restoring the non-secure CPU system registers according to PSCI specification during cold/warm boot. This is referred to as PSCI CPU Context Management. Registers that need to be preserved across CPU power down/power up cycles are maintained in cpu_context_t data structure. The initialization of other non-secure CPU system registers which do not require coordination with the EL3 Runtime Software is done directly by the PSCI library (see cm_prepare_el3_exit()).

The EL3 Runtime Software is responsible for managing register context during switch between Normal and Secure worlds. The register context to be saved and restored depends on the mechanism used to trigger the world switch. For example, if the world switch was triggered by an SMC call, then the registers need to be saved and restored according to SMCCC. In AArch64, due to the tight integration with BL31, both BL31 and PSCI library use the same cpu_context_t data structure for PSCI CPU context management and register context management during world switch. This cannot be assumed for AArch32 EL3 Runtime Software since most AArch32 Trusted OSes already implement a mechanism for register context management during world switch. Hence, when the PSCI library is integrated with a AArch32 EL3 Runtime Software, the cpu_context_t is stripped down for just PSCI CPU context management.

During cold/warm boot, after invoking appropriate PSCI library interfaces, it is expected that the EL3 Runtime Software will query the cpu_context_t and write appropriate values to the corresponding system registers. This mechanism resolves 2 additional problems for AArch32 EL3 Runtime Software:

1. Values for certain system registers like SCR and SCTLR cannot be unilaterally determined by PSCI library and need inputs from the EL3 Runtime Software. Using cpu_context_t as an intermediary data store allows EL3 Runtime Software to modify the register values appropriately before programming them.
2. The PSCI library provides appropriate LR and SPSR values (entrypoint information) for exit into non-secure world. Using cpu_context_t as an intermediary data store allows the EL3 Runtime Software to store these values safely until it is ready for exit to non-secure world.

Currently the cpu_context_t data structure for AArch32 stores the following registers: R0 - R3, LR (R14), SCR, SPSR, SCTLR.

The EL3 Runtime Software must implement accessors to get/set pointers to CPU context cpu_context_t data and these are described in *CPU Context management API*.

2.8.3 PSCI Library Interface

The PSCI library implements the PSCI Specification. The interfaces to this library are declared in psci_lib.h and are as listed below:

```
u_register_t psci_smc_handler(uint32_t smc_fid, u_register_t x1,
                              u_register_t x2, u_register_t x3,
                              u_register_t x4, void *cookie,
                              void *handle, u_register_t flags);
int psci_setup(const psci_lib_args_t *lib_args);
void psci_warmboot_entrypoint(void);
void psci_register_spd_pm_hook(const spd_pm_ops_t *pm);
void psci_prepare_next_non_secure_ctx(entry_point_info_t *next_image_info);
```

The CPU context data 'cpu_context_t' is programmed to the registers differently when PSCI is integrated with an AArch32 EL3 Runtime Software compared to when the PSCI is integrated with an AArch64 EL3 Runtime Software (BL31). For example, in the case of AArch64, there is no need to retrieve cpu_context_t data and program the registers as it will done implicitly as part of `el3_exit`. The description below of the PSCI interfaces is targeted at integration with an AArch32 EL3 Runtime Software.

The PSCI library is responsible for initializing/restoring the non-secure world to an appropriate state after boot and may choose to directly program the non-secure system registers. The PSCI generic code takes care not to directly modify any of the system registers affecting the secure world and instead returns the values to be programmed to these registers via `cpu_context_t`. The EL3 Runtime Software is responsible for programming those registers and can use the proposed values provided in the `cpu_context_t`, modifying the values if required.

PSCI library needs the flexibility to access both secure and non-secure copies of banked registers. Hence it needs to be invoked in Monitor mode for AArch32 and in EL3 for AArch64. The NS bit in SCR (in AArch32) or SCR_EL3 (in AArch64) must be set to 0. Additional requirements for the PSCI library interfaces are:

- Instruction cache must be enabled
- Both IRQ and FIQ must be masked for the current CPU
- The page tables must be setup and the MMU enabled
- The C runtime environment must be setup and stack initialized
- The Data cache must be enabled prior to invoking any of the PSCI library interfaces except for `psci_warmboot_entrypoint()`. For `psci_warmboot_entrypoint()`, if the build option `HW_ASSISTED_COHERENCY` is enabled however, data caches are expected to be enabled.

Further requirements for each interface can be found in the interface description.

Interface : psci_setup()

```
Argument : const psci_lib_args_t *lib_args
Return   : void
```

This function is to be called by the primary CPU during cold boot before any other interface to the PSCI library. It takes `lib_args`, a const pointer to `psci_lib_args_t`, as the argument. The `psci_lib_args_t` is a versioned structure and is declared in `psci_lib.h` header as follows:

```
typedef struct psci_lib_args {
    /* The version information of PSCI Library Interface */
    param_header_t        h;
    /* The warm boot entrypoint function */
    mailbox_entrypoint_t  mailbox_ep;
} psci_lib_args_t;
```

The first field `h`, of `param_header_t` type, provides the version information. The second field `mailbox_ep` is the warm boot entrypoint address and is used to configure the platform mailbox. Helper macros are provided in `psci_lib.h` to construct the `lib_args` argument statically or during runtime. Prior to calling the `psci_setup()` interface, the platform setup for cold boot must have completed. Major actions performed by this interface are:

- Initializes architecture.
- Initializes PSCI power domain and state coordination data structures.
- Calls `plat_setup_psci_ops()` with warm boot entrypoint `mailbox_ep` as argument.
- Calls `cm_set_context_by_index()` (see *CPU Context management API*) for all the CPUs in the platform

Interface : psci_prepare_next_non_secure_ctx()

```
Argument : entry_point_info_t *next_image_info
Return   : void
```

After `psci_setup()` and prior to exit to the non-secure world, this function must be called by the EL3 Runtime Software to initialize the non-secure world context. The non-secure world entrypoint information `next_image_info` (first argument) will be used to determine the non-secure context. After this function returns, the EL3 Runtime Software must retrieve the `cpu_context_t` (using cm_get_context()) for the current CPU and program the registers prior to exit to the non-secure world.

Interface : psci_register_spd_pm_hook()

```
Argument : const spd_pm_ops_t *
Return   : void
```

As explained in *Secure payload power management callback*, the EL3 Runtime Software may want to perform some bookkeeping during power management operations. This function is used to register the `spd_pm_ops_t` (first argument) callbacks with the PSCI library which will be called appropriately during power management. Calling this function is optional and need to be called by the primary CPU during the cold boot sequence after `psci_setup()` has completed.

Interface : psci_smc_handler()

```
Argument : uint32_t smc_fid, u_register_t x1,
           u_register_t x2, u_register_t x3,
           u_register_t x4, void *cookie,
           void *handle, u_register_t flags
Return   : u_register_t
```

This function is the top level handler for SMCs which fall within the PSCI service range specified in SMCCC. The function ID `smc_fid` (first argument) determines the PSCI API to be called. The `x1` to `x4` (2nd to 5th arguments), are the values of the registers r1 - r4 (in AArch32) or x1 - x4 (in AArch64) when the SMC is received. These are the arguments to PSCI API as described in PSCI spec. The 'flags' (8th argument) is a bit field parameter and is detailed in 'smccc.h' header. It includes whether the call is from the secure or non-secure world. The `cookie` (6th argument) and the `handle` (7th argument) are not used and are reserved for future use.

The return value from this interface is the return value from the underlying PSCI API corresponding to `smc_fid`. This function may not return back to the caller if PSCI API causes power down of the CPU. In this case, when the CPU wakes up, it will start execution from the warm reset address.

Interface : psci_warmboot_entrypoint()

```
Argument : void
Return   : void
```

This function performs the warm boot initialization/restoration as mandated by PSCI spec. For AArch32, on wakeup from power down the CPU resets to secure SVC mode and the EL3 Runtime Software must perform the prerequisite initializations mentioned at top of this section. This function must be called with Data cache disabled (unless build option `HW_ASSISTED_COHERENCY` is enabled) but with MMU initialized and enabled. The major actions performed by this function are:

- Invalidates the stack and enables the data cache.

- Initializes architecture and PSCI state coordination.
- Restores/Initializes the peripheral drivers to the required state via appropriate `plat_psci_ops_t` hooks
- Restores the EL3 Runtime Software context via appropriate `spd_pm_ops_t` callbacks.
- Restores/Initializes the non-secure context and populates the `cpu_context_t` for the current CPU.

Upon the return of this function, the EL3 Runtime Software must retrieve the non-secure `cpu_context_t` using `cm_get_context()` and program the registers prior to exit to the non-secure world.

2.8.4 EL3 Runtime Software dependencies

The PSCI Library includes supporting frameworks like context management, cpu operations (cpu_ops) and per-cpu data framework. Other helper library functions like bakery locks and spin locks are also included in the library. The dependencies which must be fulfilled by the EL3 Runtime Software for integration with PSCI library are described below.

General dependencies

The PSCI library being a Multiprocessor (MP) implementation, EL3 Runtime Software must provide an SMC handling framework capable of MP adhering to SMCCC specification.

The EL3 Runtime Software must also export cache maintenance primitives and some helper utilities for assert, print and memory operations as listed below. The TF-A source tree provides implementations for all these functions but the EL3 Runtime Software may use its own implementation.

Functions : assert(), memcpy(), memset(), printf()

These must be implemented as described in ISO C Standard.

Function : flush_dcache_range()

```
Argument : uintptr_t addr, size_t size
Return   : void
```

This function cleans and invalidates (flushes) the data cache for memory at address `addr` (first argument) address and of size `size` (second argument).

Function : inv_dcache_range()

```
Argument : uintptr_t addr, size_t size
Return   : void
```

This function invalidates (flushes) the data cache for memory at address `addr` (first argument) address and of size `size` (second argument).

Function : do_panic()

```
Argument : void
Return   : void
```

This function will be called by the PSCI library on encountering a critical failure that cannot be recovered from. This function **must not** return.

CPU Context management API

The CPU context management data memory is statically allocated by PSCI library in BSS section. The PSCI library requires the EL3 Runtime Software to implement APIs to store and retrieve pointers to this CPU context data. SP-MIN demonstrates how these APIs can be implemented but the EL3 Runtime Software can choose a more optimal implementation (like dedicating the secure TPIDRPRW system register (in AArch32) for storing these pointers).

Function : cm_set_context_by_index()

```
Argument : unsigned int cpu_idx, void *context, unsigned int security_state
Return   : void
```

This function is called during cold boot when the `psci_setup()` PSCI library interface is called.

This function must store the pointer to the CPU context data, `context` (2nd argument), for the specified `security_state` (3rd argument) and CPU identified by `cpu_idx` (first argument). The `security_state` will always be non-secure when called by PSCI library and this argument is retained for compatibility with BL31. The `cpu_idx` will correspond to the index returned by the `plat_core_pos_by_mpidr()` for `mpidr` of the CPU.

The actual method of storing the `context` pointers is implementation specific. For example, SP-MIN stores the pointers in the array `sp_min_cpu_ctx_ptr` declared in `sp_min_main.c`.

Function : cm_get_context()

```
Argument : uint32_t security_state
Return   : void *
```

This function must return the pointer to the `cpu_context_t` structure for the specified `security_state` (first argument) for the current CPU. The caller must ensure that `cm_set_context_by_index` is called first and the appropriate context pointers are stored prior to invoking this API. The `security_state` will always be non-secure when called by PSCI library and this argument is retained for compatibility with BL31.

Function : cm_get_context_by_index()

```
Argument : unsigned int cpu_idx, unsigned int security_state
Return   : void *
```

This function must return the pointer to the `cpu_context_t` structure for the specified `security_state` (second argument) for the CPU identified by `cpu_idx` (first argument). The caller must ensure that `cm_set_context_by_index` is called first and the appropriate context pointers are stored prior to invoking this API. The `security_state` will always be non-secure when called by PSCI library and this argument is retained for compatibility with BL31. The `cpu_idx` will correspond to the index returned by the `plat_core_pos_by_mpidr()` for `mpidr` of the CPU.

Platform API

The platform layer abstracts the platform-specific details from the generic PSCI library. The following platform APIs/macros must be defined by the EL3 Runtime Software for integration with the PSCI library.

The mandatory platform APIs are:

- plat_my_core_pos
- plat_core_pos_by_mpidr
- plat_get_syscnt_freq2
- plat_get_power_domain_tree_desc
- plat_setup_psci_ops

- plat_reset_handler
- plat_panic_handler
- plat_get_my_stack

The mandatory platform macros are:

- PLATFORM_CORE_COUNT
- PLAT_MAX_PWR_LVL
- PLAT_NUM_PWR_DOMAINS
- CACHE_WRITEBACK_GRANULE
- PLAT_MAX_OFF_STATE
- PLAT_MAX_RET_STATE
- PLAT_MAX_PWR_LVL_STATES (optional)
- PLAT_PCPU_DATA_SIZE (optional)

The details of these APIs/macros can be found in the *Porting Guide*.

All platform specific operations for power management are done via `plat_psci_ops_t` callbacks registered by the platform when `plat_setup_psci_ops()` API is called. The description of each of the callbacks in `plat_psci_ops_t` can be found in PSCI section of the *Porting Guide*. If any these callbacks are not registered, then the PSCI API associated with that callback will not be supported by PSCI library.

Secure payload power management callback

During PSCI power management operations, the EL3 Runtime Software may need to perform some bookkeeping, and PSCI library provides `spd_pm_ops_t` callbacks for this purpose. These hooks must be populated and registered by using `psci_register_spd_pm_hook()` PSCI library interface.

Typical bookkeeping during PSCI power management calls include save/restore of the EL3 Runtime Software context. Also if the EL3 Runtime Software makes use of secure interrupts, then these interrupts must also be managed appropriately during CPU power down/power up. Any secure interrupt targeted to the current CPU must be disabled or re-targeted to other running CPU prior to power down of the current CPU. During power up, these interrupt can be enabled/re-targeted back to the current CPU.

```
typedef struct spd_pm_ops {
        void (*svc_on)(u_register_t target_cpu);
        int32_t (*svc_off)(u_register_t __unused);
        void (*svc_suspend)(u_register_t max_off_pwrlvl);
        void (*svc_on_finish)(u_register_t __unused);
        void (*svc_suspend_finish)(u_register_t max_off_pwrlvl);
        int32_t (*svc_migrate)(u_register_t from_cpu, u_register_t to_cpu);
        int32_t (*svc_migrate_info)(u_register_t *resident_cpu);
        void (*svc_system_off)(void);
        void (*svc_system_reset)(void);
} spd_pm_ops_t;
```

A brief description of each callback is given below:

- svc_on, svc_off, svc_on_finish

  The `svc_on`, `svc_off` callbacks are called during PSCI_CPU_ON, PSCI_CPU_OFF APIs respectively. The `svc_on_finish` is called when the target CPU of PSCI_CPU_ON API powers up and executes the `psci_warmboot_entrypoint()` PSCI library interface.

- svc_suspend, svc_suspend_finish

  The `svc_suspend` callback is called during power down bu either PSCI_SUSPEND or PSCI_SYSTEM_SUSPEND APIs. The `svc_suspend_finish` is called when the CPU wakes up from suspend and executes the `psci_warmboot_entrypoint()` PSCI library interface. The `max_off_pwrlvl` (first parameter) denotes the highest power domain level being powered down to or woken up from suspend.

- svc_system_off, svc_system_reset

  These callbacks are called during PSCI_SYSTEM_OFF and PSCI_SYSTEM_RESET PSCI APIs respectively.

- svc_migrate_info

  This callback is called in response to PSCI_MIGRATE_INFO_TYPE or PSCI_MIGRATE_INFO_UP_CPU APIs. The return value of this callback must correspond to the return value of PSCI_MIGRATE_INFO_TYPE API as described in PSCI spec. If the secure payload is a Uniprocessor (UP) implementation, then it must update the mpidr of the CPU it is resident in via `resident_cpu` (first argument). The updates to `resident_cpu` is ignored if the secure payload is a multiprocessor (MP) implementation.

- svc_migrate

  This callback is only relevant if the secure payload in EL3 Runtime Software is a Uniprocessor (UP) implementation and supports migration from the current CPU `from_cpu` (first argument) to another CPU `to_cpu` (second argument). This callback is called in response to PSCI_MIGRATE API. This callback is never called if the secure payload is a Multiprocessor (MP) implementation.

CPU operations

The CPU operations (cpu_ops) framework implement power down sequence specific to the CPU and the details of which can be found at *CPU specific operations framework*. The TF-A tree implements the `cpu_ops` for various supported CPUs and the EL3 Runtime Software needs to include the required `cpu_ops` in its build. The start and end of the `cpu_ops` descriptors must be exported by the EL3 Runtime Software via the `__CPU_OPS_START__` and `__CPU_OPS_END__` linker symbols.

The `cpu_ops` descriptors also include reset sequences and may include errata workarounds for the CPU. The EL3 Runtime Software can choose to call this during cold/warm reset if it does not implement its own reset sequence/errata workarounds.

2.9 EL3 Runtime Service Writer's Guide

2.9.1 Introduction

This document describes how to add a runtime service to the EL3 Runtime Firmware component of Trusted Firmware-A (TF-A), BL31.

Software executing in the normal world and in the trusted world at exception levels lower than EL3 will request runtime services using the Secure Monitor Call (SMC) instruction. These requests will follow the convention described in the SMC Calling Convention PDD (SMCCC). The SMCCC assigns function identifiers to each SMC request and describes how arguments are passed and results are returned.

SMC Functions are grouped together based on the implementor of the service, for example a subset of the Function IDs are designated as “OEM Calls” (see SMCCC for full details). The EL3 runtime services framework in BL31 enables the independent implementation of services for each group, which are then compiled into the BL31 image.

This simplifies the integration of common software from Arm to support PSCI, Secure Monitor for a Trusted OS and SoC specific software. The common runtime services framework ensures that SMC Functions are dispatched to their respective service implementation - the *Firmware Design* document provides details of how this is achieved.

The interface and operation of the runtime services depends heavily on the concepts and definitions described in the SMCCC, in particular SMC Function IDs, Owning Entity Numbers (OEN), Fast and Standard calls, and the SMC32 and SMC64 calling conventions. Please refer to that document for a full explanation of these terms.

2.9.2 Owning Entities, Call Types and Function IDs

The SMC Function Identifier includes a OEN field. These values and their meaning are described in SMCCC and summarized in table 1 below. Some entities are allocated a range of of OENs. The OEN must be interpreted in conjunction with the SMC call type, which is either *Fast* or *Yielding*. Fast calls are uninterruptible whereas Yielding calls can be pre-empted. The majority of Owning Entities only have allocated ranges for Fast calls: Yielding calls are reserved exclusively for Trusted OS providers or for interoperability with legacy 32-bit software that predates the SMCCC.

```
Type       OEN     Service
Fast        0      Arm Architecture calls
Fast        1      CPU Service calls
Fast        2      SiP Service calls
Fast        3      OEM Service calls
Fast        4      Standard Service calls
Fast       5-47    Reserved for future use
Fast      48-49    Trusted Application calls
Fast      50-63    Trusted OS calls

Yielding   0- 1    Reserved for existing Armv7-A calls
Yielding   2-63    Trusted OS Standard Calls
```

Table 1: Service types and their corresponding Owning Entity Numbers

Each individual entity can allocate the valid identifiers within the entity range as they need - it is not necessary to coordinate with other entities of the same type. For example, two SoC providers can use the same Function ID within the SiP Service calls OEN range to mean different things - as these calls should be specific to the SoC. The Standard Runtime Calls OEN is used for services defined by Arm standards, such as PSCI.

The SMC Function ID also indicates whether the call has followed the SMC32 calling convention, where all parameters are 32-bit, or the SMC64 calling convention, where the parameters are 64-bit. The framework identifies and rejects invalid calls that use the SMC64 calling convention but that originate from an AArch32 caller.

The EL3 runtime services framework uses the call type and OEN to identify a specific handler for each SMC call, but it is expected that an individual handler will be responsible for all SMC Functions within a given service type.

2.9.3 Getting started

TF-A has a `services` directory in the source tree under which each owning entity can place the implementation of its runtime service. The PSCI implementation is located here in the `lib/psci` directory.

Runtime service sources will need to include the `runtime_svc.h` header file.

2.9.4 Registering a runtime service

A runtime service is registered using the `DECLARE_RT_SVC()` macro, specifying the name of the service, the range of OENs covered, the type of service and initialization and call handler functions.

```
#define DECLARE_RT_SVC(_name, _start, _end, _type, _setup, _smch)
```

- `_name` is used to identify the data structure declared by this macro, and is also used for diagnostic purposes
- `_start` and `_end` values must be based on the `OEN_*` values defined in `smccc.h`
- `_type` must be one of `SMC_TYPE_FAST` or `SMC_TYPE_YIELD`
- `_setup` is the initialization function with the `rt_svc_init` signature:

  ```
  typedef int32_t (*rt_svc_init)(void);
  ```

- `_smch` is the SMC handler function with the `rt_svc_handle` signature:

  ```
  typedef uintptr_t (*rt_svc_handle_t)(uint32_t smc_fid,
                                       u_register_t x1, u_register_t x2,
                                       u_register_t x3, u_register_t x4,
                                       void *cookie,
                                       void *handle,
                                       u_register_t flags);
  ```

Details of the requirements and behavior of the two callbacks is provided in the following sections.

During initialization the services framework validates each declared service to ensure that the following conditions are met:

1. The `_start` OEN is not greater than the `_end` OEN
2. The `_end` OEN does not exceed the maximum OEN value (63)
3. The `_type` is one of `SMC_TYPE_FAST` or `SMC_TYPE_YIELD`
4. `_setup` and `_smch` routines have been specified

`std_svc_setup.c` provides an example of registering a runtime service:

```
/* Register Standard Service Calls as runtime service */
DECLARE_RT_SVC(
        std_svc,
        OEN_STD_START,
        OEN_STD_END,
        SMC_TYPE_FAST,
        std_svc_setup,
        std_svc_smc_handler
);
```

2.9.5 Initializing a runtime service

Runtime services are initialized once, during cold boot, by the primary CPU after platform and architectural initialization is complete. The framework performs basic validation of the declared service before calling the service initialization function (`_setup` in the declaration). This function must carry out any essential EL3 initialization prior to receiving a SMC Function call via the handler function.

On success, the initialization function must return `0`. Any other return value will cause the framework to issue a diagnostic:

```
Error initializing runtime service <name of the service>
```

and then ignore the service - the system will continue to boot but SMC calls will not be passed to the service handler and instead return the *Unknown SMC Function ID* result `0xFFFFFFFF`.

If the system must not be allowed to proceed without the service, the initialization function must itself cause the firmware boot to be halted.

If the service uses per-CPU data this must either be initialized for all CPUs during this call, or be done lazily when a CPU first issues an SMC call to that service.

2.9.6 Handling runtime service requests

SMC calls for a service are forwarded by the framework to the service's SMC handler function (`_smch` in the service declaration). This function must have the following signature:

```
typedef uintptr_t (*rt_svc_handle_t)(uint32_t smc_fid,
                                     u_register_t x1, u_register_t x2,
                                     u_register_t x3, u_register_t x4,
                                     void *cookie,
                                     void *handle,
                                     u_register_t flags);
```

The handler is responsible for:

1. Determining that `smc_fid` is a valid and supported SMC Function ID, otherwise completing the request with the *Unknown SMC Function ID*:

   ```
   SMC_RET1(handle, SMC_UNK);
   ```

2. Determining if the requested function is valid for the calling security state. SMC Calls can be made from both the normal and trusted worlds and the framework will forward all calls to the service handler.

   The `flags` parameter to this function indicates the caller security state in bit[0], where a value of `1` indicates a non-secure caller. The `is_caller_secure(flags)` and `is_caller_non_secure(flags)` can be used to test this condition.

   If invalid, the request should be completed with:

   ```
   SMC_RET1(handle, SMC_UNK);
   ```

3. Truncating parameters for calls made using the SMC32 calling convention. Such calls can be determined by checking the CC field in bit[30] of the `smc_fid` parameter, for example by using:

   ```
   if (GET_SMC_CC(smc_fid) == SMC_32) ...
   ```

For such calls, the upper bits of the parameters x1-x4 and the saved parameters X5-X7 are UNDEFINED and must be explicitly ignored by the handler. This can be done by truncating the values to a suitable 32-bit integer type before use, for example by ensuring that functions defined to handle individual SMC Functions use appropriate 32-bit parameters.

4. Providing the service requested by the SMC Function, utilizing the immediate parameters x1-x4 and/or the additional saved parameters X5-X7. The latter can be retrieved using the `SMC_GET_GP(handle, ref)` function, supplying the appropriate `CTX_GPREG_Xn` reference, e.g.

```
uint64_t x6 = SMC_GET_GP(handle, CTX_GPREG_X6);
```

5. Implementing the standard SMC32 Functions that provide information about the implementation of the service. These are the Call Count, Implementor UID and Revision Details for each service documented in section 6 of the SMCCC.

   TF-A expects owning entities to follow this recommendation.

6. Returning the result to the caller. Based on SMCCC spec, results are returned in W0-W7(X0-X7) registers for SMC32(SMC64) calls from AArch64 state. Results are returned in R0-R7 registers for SMC32 calls from AArch32 state. The framework provides a family of macros to set the multi-register return value and complete the handler:

```
AArch64 state:

SMC_RET1(handle, x0);
SMC_RET2(handle, x0, x1);
SMC_RET3(handle, x0, x1, x2);
SMC_RET4(handle, x0, x1, x2, x3);
SMC_RET5(handle, x0, x1, x2, x3, x4);
SMC_RET6(handle, x0, x1, x2, x3, x4, x5);
SMC_RET7(handle, x0, x1, x2, x3, x4, x5, x6);
SMC_RET8(handle, x0, x1, x2, x3, x4, x5, x6, x7);

AArch32 state:

SMC_RET1(handle, r0);
SMC_RET2(handle, r0, r1);
SMC_RET3(handle, r0, r1, r2);
SMC_RET4(handle, r0, r1, r2, r3);
SMC_RET5(handle, r0, r1, r2, r3, r4);
SMC_RET6(handle, r0, r1, r2, r3, r4, r5);
SMC_RET7(handle, r0, r1, r2, r3, r4, r5, r6);
SMC_RET8(handle, r0, r1, r2, r3, r4, r5, r6, r7);
```

The `cookie` parameter to the handler is reserved for future use and can be ignored. The `handle` is returned by the SMC handler - completion of the handler function must always be via one of the `SMC_RETn()` macros.

Note: The PSCI and Test Secure-EL1 Payload Dispatcher services do not follow all of the above requirements yet.

2.9.7 Services that contain multiple sub-services

It is possible that a single owning entity implements multiple sub-services. For example, the Standard calls service handles `0x84000000-0x8400FFFF` and `0xC4000000-0xC400FFFF` functions. Within that range, the PSCI service handles the `0x84000000-0x8400001F` and `0xC4000000-0xC400001F` functions. In that respect, PSCI is a 'sub-service' of the Standard calls service. In future, there could be additional such sub-services in the Standard calls service which perform independent functions.

In this situation it may be valuable to introduce a second level framework to enable independent implementation of sub-services. Such a framework might look very similar to the current runtime services framework, but using a different part of the SMC Function ID to identify the sub-service. TF-A does not provide such a framework at present.

2.9.8 Secure-EL1 Payload Dispatcher service (SPD)

Services that handle SMC Functions targeting a Trusted OS, Trusted Application, or other Secure-EL1 Payload are special. These services need to manage the Secure-EL1 context, provide the *Secure Monitor* functionality of switching between the normal and secure worlds, deliver SMC Calls through to Secure-EL1 and generally manage the Secure-EL1 Payload through CPU power-state transitions.

TODO: Provide details of the additional work required to implement a SPD and the BL31 support for these services. Or a reference to the document that will provide this information. . . .

CHAPTER

THREE

PROCESSES & POLICIES

3.1 Security Handling

3.1.1 Security Disclosures

We disclose all security vulnerabilities we find, or are advised about, that are relevant to Trusted Firmware-A. We encourage responsible disclosure of vulnerabilities and inform users as best we can about all possible issues.

We disclose TF-A vulnerabilities as Security Advisories, all of which are listed at the bottom of this page. Any new ones will, additionally, be announced as issues in the project's issue tracker with the `security-advisory` tag. You can receive notification emails for these by watching the "Trusted Firmware-A" project at https://developer.trustedfirmware.org/.

3.1.2 Found a Security Issue?

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

Warning: If you think you have found a security vulnerability, please **do not** report it in the issue tracker or on the mailing list. Instead, please follow the TrustedFirmware.org security incident process.

One of the goals of this process is to ensure providers of products that use TF-A 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 process. We do our best to respond and fix any issues quickly.

Afterwards, we encourage you to write-up your findings about the TF-A source code.

3.1.3 Attribution

We will name and thank you in the *Change Log & Release Notes* distributed with the source code and in any published security advisory.

3.1.4 Security Advisories

ID	Title
Advisory TFV-1 (CVE-2016-10319)	Malformed Firmware Update SMC can result in copy of unexpectedly large data into secure memory
Advisory TFV-2 (CVE-2017-7564)	Enabled secure self-hosted invasive debug interface can allow normal world to panic secure world
Advisory TFV-3 (CVE-2017-7563)	RO memory is always executable at AArch64 Secure EL1
Advisory TFV-4 (CVE-2017-9607)	Malformed Firmware Update SMC can result in copy or authentication of unexpected data in secure memory in AArch32 state
Advisory TFV-5 (CVE-2017-15031)	Not initializing or saving/restoring PMCR_EL0 can leak secure world timing information
Advisory TFV-6 (CVE-2017-5753, CVE-2017-5715, CVE-2017-5754)	Trusted Firmware-A exposure to speculative processor vulnerabilities using cache timing side-channels
Advisory TFV-7 (CVE-2018-3639)	Trusted Firmware-A exposure to cache speculation vulnerability Variant 4
Advisory TFV-8 (CVE-2018-19440)	Not saving x0 to x3 registers can leak information from one Normal World SMC client to another

---

3.2 Platform Compatibility Policy

3.2.1 Introduction

This document clarifies the project's policy around compatibility for upstream platforms.

3.2.2 Platform compatibility policy

Platform compatibility is mainly affected by changes to Platform APIs (as documented in the *Porting Guide*), driver APIs (like the GICv3 drivers) or library interfaces (like xlat_table library). The project will try to maintain compatibility for upstream platforms. Due to evolving requirements and enhancements, there might be changes affecting platform compatibility which means the previous interface needs to be deprecated and a new interface introduced to replace it. In case the migration to the new interface is trivial, the contributor of the change is expected to make good effort to migrate the upstream platforms to the new interface.

The deprecated interfaces are listed inside *Release Processes* as well as the release after which each one will be removed. When an interface is deprecated, the page must be updated to indicate the release after which the interface will be removed. This must be at least 1 full release cycle in future. For non-trivial interface changes, an email should be sent out to the TF-A public mailing list to notify platforms that they should migrate away from the deprecated interfaces. Platforms are expected to migrate before the removal of the deprecated interface.

3.3 Coding Style

The following sections outline the *TF-A* coding style for *C* code. The style is based on the Linux kernel coding style, with a few modifications.

The style should not be considered *set in stone*. Feel free to provide feedback and suggestions.

Note: You will almost certainly find code in the *TF-A* repository that does not follow the style. The intent is for all code to do so eventually.

3.3.1 File Encoding

The source code must use the **UTF-8** character encoding. Comments and documentation may use non-ASCII characters when required (e.g. Greek letters used for units) but code itself is still limited to ASCII characters.

Newlines must be in **Unix** style, which means that only the Line Feed (`LF`) character is used to break a line and reset to the first column.

3.3.2 Language

The primary language for comments and naming must be International English. In cases where there is a conflict between the American English and British English spellings of a word, the American English spelling is used.

Exceptions are made when referring directly to something that does not use international style, such as the name of a company. In these cases the existing name should be used as-is.

3.3.3 C Language Standard

The C language mode used for TF-A is *GNU99*. This is the "GNU dialect of ISO C99", which implies the *ISO C99* standard with GNU extensions.

Both GCC and Clang compiler toolchains have support for *GNU99* mode, though Clang does lack support for a small number of GNU extensions. These missing extensions are rarely used, however, and should not pose a problem.

3.3.4 MISRA Compliance

TF-A attempts to comply with the MISRA C:2012 Guidelines. Coverity Static Analysis is used to regularly generate a report of current MISRA defects and to prevent the addition of new ones.

It is not possible for the project to follow all MISRA guidelines. We maintain a spreadsheet that lists all rules and directives and whether we aim to comply with them or not. A rationale is given for each deviation.

Note: Enforcing a rule does not mean that the codebase is free of defects of that rule, only that they would ideally be removed.

Note: Third-party libraries are not considered in our MISRA analysis and we do not intend to modify them to make them MISRA compliant.

3.3.5 Indentation

Use **tabs** for indentation. The use of spaces for indentation is forbidden except in the case where a term is being indented to a boundary that cannot be achieved using tabs alone.

Tab spacing should be set to **8 characters**.

Trailing whitespace is not allowed and must be trimmed.

3.3.6 Spacing

Single spacing should be used around most operators, including:

- Arithmetic operators (`+`, `-`, `/`, `*`)
- Assignment operators (`=`, `+=`, etc)
- Boolean operators (`&&`, `||`)
- Comparison operators (`<`, `>`, `==`, etc)

A space should also be used to separate parentheses and braces when they are not already separated by a newline, such as for the `if` statement in the following example:

```
int function_foo(bool bar)
{
    if (bar) {
        function_baz();
    }
}
```

Note that there is no space between the name of a function and the following parentheses.

Control statements (`if`, `for`, `switch`, `while`, etc) must be separated from the following open paranthesis by a single space. The previous example illustrates this for an `if` statement.

3.3.7 Line Length

Line length *should* be at most **80 characters**. This limit does not include non-printing characters such as the line feed.

This rule is a *should*, not a must, and it is acceptable to exceed the limit **slightly** where the readability of the code would otherwise be significantly reduced. Use your judgement in these cases.

3.3.8 Blank Lines

Functions are usually separated by a single blank line. In certain cases it is acceptable to use additional blank lines for clarity, if required.

The file must end with a single newline character. Many editors have the option to insert this automatically and to trim multiple blank lines at the end of the file.

3.3.9 Braces

Opening Brace Placement

Braces follow the **Kernighan and Ritchie (K&R)** style, where the opening brace is **not** placed on a new line.

Example for a `while` loop:

```
while (condition) {
    foo();
    bar();
}
```

This style applies to all blocks except for functions which, following the Linux style, **do** place the opening brace on a new line.

Example for a function:

```
int my_function(void)
{
    int a;

    a = 1;
    return a;
}
```

Conditional Statement Bodies

Where conditional statements (such as `if`, `for`, `while` and `do`) are used, braces must be placed around the statements that form the body of the conditional. This is the case regardless of the number of statements in the body.

Note: This is a notable departure from the Linux coding style that has been adopted to follow MISRA guidelines more closely and to help prevent errors.

For example, use the following style:

```
if (condition) {
    foo++;
}
```

instead of omitting the optional braces around a single statement:

```
/* This is violating MISRA C 2012: Rule 15.6 */
if (condition)
    foo++;
```

The reason for this is to prevent accidental changes to control flow when modifying the body of the conditional. For example, at a quick glance it is easy to think that the value of `bar` is only incremented if `condition` evaluates to `true` but this is not the case - `bar` will always be incremented regardless of the condition evaluation. If the developer forgets to add braces around the conditional body when adding the `bar++;` statement then the program execution will not proceed as intended.

```
/* This is violating MISRA C 2012: Rule 15.6 */
if (condition)
```

(continues on next page)

(continued from previous page)

```
    foo++;
    bar++;
```

3.3.10 Naming

Functions

Use lowercase for function names, separating multiple words with an underscore character (_). This is sometimes referred to as *Snake Case*. An example is given below:

```
void bl2_arch_setup(void)
{
    ...
}
```

Local Variables and Parameters

Local variables and function parameters use the same format as function names: lowercase with underscore separation between multiple words. An example is given below:

```
static void set_scr_el3_from_rm(uint32_t type,
                                uint32_t interrupt_type_flags,
                                uint32_t security_state)
{
    uint32_t flag, bit_pos;

    ...

}
```

Preprocessor Macros

Identifiers that are defined using preprocessor macros are written in all uppercase text.

```
#define BUFFER_SIZE_BYTES 64
```

3.3.11 Function Attributes

Place any function attributes after the function type and before the function name.

```
void __init plat_arm_interconnect_init(void);
```

3.3.12 Alignment

Alignment should be performed primarily with tabs, adding spaces if required to achieve a granularity that is smaller than the tab size. For example, with a tab size of eight columns it would be necessary to use one tab character and two spaces to indent text by ten columns.

Switch Statement Alignment

When using `switch` statements, align each `case` statement with the `switch` so that they are in the same column.

```
switch (condition) {
case A:
    foo();
case B:
    bar();
default:
    baz();
}
```

Pointer Alignment

The reference and dereference operators (ampersand and *pointer star*) must be aligned with the name of the object on which they are operating, as opposed to the type of the object.

```
uint8_t *foo;

foo = &bar;
```

3.3.13 Comments

The general rule for comments is that the double-slash style of comment (`//`) is not allowed. Examples of the allowed comment formats are shown below:

```
/*
 * This example illustrates the first allowed style for multi-line comments.
 *
 * Blank lines within multi-lines are allowed when they add clarity or when
 * they separate multiple contexts.
 *
 */
```

```
/**************************************************************************
 * This is the second allowed style for multi-line comments.
 *
 * In this style, the first and last lines use asterisks that run the full
 * width of the comment at its widest point.
 *
 * This style can be used for additional emphasis.
 *
 *************************************************************************/
```

```
/* Single line comments can use this format */
```

```
/*****************************************************************************
 * This alternative single-line comment style can also be used for emphasis.
 *****************************************************************************/
```

3.3.14 Headers and inclusion

Header guards

For a header file called “some_driver.h” the style used by *TF-A* is:

```
#ifndef SOME_DRIVER_H
#define SOME_DRIVER_H

<header content>

#endif /* SOME_DRIVER_H */
```

Include statement ordering

All header files that are included by a source file must use the following, grouped ordering. This is to improve readability (by making it easier to quickly read through the list of headers) and maintainability.

1. *System* includes: Header files from the standard *C* library, such as `stddef.h` and `string.h`.
2. *Project* includes: Header files under the `include/` directory within *TF-A* are *project* includes.
3. *Platform* includes: Header files relating to a single, specific platform, and which are located under the `plat/<platform_name>` directory within *TF-A*, are *platform* includes.

Within each group, `#include` statements must be in alphabetical order, taking both the file and directory names into account.

Groups must be separated by a single blank line for clarity.

The example below illustrates the ordering rules using some contrived header file names; this type of name reuse should be otherwise avoided.

```
#include <string.h>

#include <a_dir/example/a_header.h>
#include <a_dir/example/b_header.h>
#include <a_dir/test/a_header.h>
#include <b_dir/example/a_header.h>

#include "a_header.h"
```

Include statement variants

Two variants of the `#include` directive are acceptable in the *TF-A* codebase. Correct use of the two styles improves readability by suggesting the location of the included header and reducing ambiguity in cases where generic and platform-specific headers share a name.

For header files that are in the same directory as the source file that is including them, use the `"..."` variant.

For header files that are **not** in the same directory as the source file that is including them, use the `<...>` variant.

Example (bl1_fwu.c):

```
#include <assert.h>
#include <errno.h>
#include <string.h>

#include "bl1_private.h"
```

3.3.15 Typedefs

Avoid anonymous typedefs of structs/enums in headers

For example, the following definition:

```
typedef struct {
        int arg1;
        int arg2;
} my_struct_t;
```

is better written as:

```
struct my_struct {
        int arg1;
        int arg2;
};
```

This allows function declarations in other header files that depend on the struct/enum to forward declare the struct/enum instead of including the entire header:

```
struct my_struct;
void my_func(struct my_struct *arg);
```

instead of:

```
#include <my_struct.h>
void my_func(my_struct_t *arg);
```

Some TF definitions use both a struct/enum name **and** a typedef name. This is discouraged for new definitions as it makes it difficult for TF to comply with MISRA rule 8.3, which states that “All declarations of an object or function shall use the same names and type qualifiers”.

The Linux coding standards also discourage new typedefs and checkpatch emits a warning for this.

Existing typedefs will be retained for compatibility.

3.4 Coding Guidelines

This document provides some additional guidelines to consider when writing *TF-A* code. These are not intended to be strictly-enforced rules like the contents of the *Coding Style*.

3.4.1 Automatic Editor Configuration

Many of the rules given below (such as indentation size, use of tabs, and newlines) can be set automatically using the EditorConfig configuration file in the root of the repository: `.editorconfig`. With a supported editor, the rules set out in this file can be automatically applied when you are editing files in the *TF-A* repository.

Several editors include built-in support for EditorConfig files, and many others support its functionality through plug-ins.

Use of the EditorConfig file is suggested but is not required.

3.4.2 Automatic Compliance Checking

To assist with coding style compliance, the project Makefile contains two targets which both utilise the *checkpatch.pl* script that ships with the Linux source tree. The project also defines certain *checkpatch* options in the `.checkpatch.conf` file in the top-level directory.

Note: Checkpatch errors will gate upstream merging of pull requests. Checkpatch warnings will not gate merging but should be reviewed and fixed if possible.

To check the entire source tree, you must first download copies of `checkpatch.pl`, `spelling.txt` and `const_structs.checkpatch` available in the Linux master tree *scripts* directory, then set the `CHECKPATCH` environment variable to point to `checkpatch.pl` (with the other 2 files in the same directory) and build the *checkcodebase* target:

```
make CHECKPATCH=<path-to-linux>/linux/scripts/checkpatch.pl checkcodebase
```

To just check the style on the files that differ between your local branch and the remote master, use:

```
make CHECKPATCH=<path-to-linux>/linux/scripts/checkpatch.pl checkpatch
```

If you wish to check your patch against something other than the remote master, set the `BASE_COMMIT` variable to your desired branch. By default, `BASE_COMMIT` is set to `origin/master`.

Ignored Checkpatch Warnings

Some checkpatch warnings in the TF codebase are deliberately ignored. These include:

- `**WARNING: line over 80 characters**`: Although the codebase should generally conform to the 80 character limit this is overly restrictive in some cases.
- `**WARNING: Use of volatile is usually wrong`: see Why the "volatile" type class should not be used . Although this document contains some very useful information, there are several legimate uses of the volatile keyword within the TF codebase.

3.4.3 Performance considerations

Avoid printf and use logging macros

`debug.h` provides logging macros (for example, `WARN` and `ERROR`) which wrap `tf_log` and which allow the logging call to be compiled-out depending on the `make` command. Use these macros to avoid print statements being compiled unconditionally into the binary.

Each logging macro has a numerical log level:

```
#define LOG_LEVEL_NONE     0
#define LOG_LEVEL_ERROR    10
#define LOG_LEVEL_NOTICE   20
#define LOG_LEVEL_WARNING  30
#define LOG_LEVEL_INFO     40
#define LOG_LEVEL_VERBOSE  50
```

By default, all logging statements with a log level `<= LOG_LEVEL_INFO` will be compiled into debug builds and all statements with a log level `<= LOG_LEVEL_NOTICE` will be compiled into release builds. This can be overridden from the command line or by the platform makefile (although it may be necessary to clean the build directory first).

For example, to enable `VERBOSE` logging on FVP:

```
make PLAT=fvp LOG_LEVEL=50 all
```

Use const data where possible

For example, the following code:

```
struct my_struct {
        int arg1;
        int arg2;
};

void init(struct my_struct *ptr);

void main(void)
{
        struct my_struct x;
        x.arg1 = 1;
        x.arg2 = 2;
        init(&x);
}
```

is better written as:

```
struct my_struct {
        int arg1;
        int arg2;
};

void init(const struct my_struct *ptr);

void main(void)
{
        const struct my_struct x = { 1, 2 };
```

(continues on next page)

(continued from previous page)

```
        init(&x);
}
```

This allows the linker to put the data in a read-only data section instead of a writeable data section, which may result in a smaller and faster binary. Note that this may require dependent functions (`init()` in the above example) to have `const` arguments, assuming they don't need to modify the data.

3.4.4 Libc functions that are banned or to be used with caution

Below is a list of functions that present security risks and either must not be used (Banned) or are discouraged from use and must be used with care (Caution).

libc function	Status	Comments
`strcpy, wcscpy, strncpy`	Banned	use strlcpy instead
`strcat, wcscat, strncat`	Banned	use strlcat instead
`sprintf, vsprintf`	Banned	use snprintf, vsnprintf instead
`snprintf`	Caution	ensure result fits in buffer i.e : snprintf(buf,size. . .) < size
`vsnprintf`	Caution	inspect va_list match types specified in format string
`strtok`	Banned	use strtok_r or strsep instead
`strtok_r, strsep`	Caution	inspect for terminated input buffer
`ato*`	Banned	use equivalent strto* functions
`*toa`	Banned	Use snprintf instead

The *libc* component in the codebase will not add support for the banned APIs.

3.4.5 Error handling and robustness

Using CASSERT to check for compile time data errors

Where possible, use the `CASSERT` macro to check the validity of data known at compile time instead of checking validity at runtime, to avoid unnecessary runtime code.

For example, this can be used to check that the assembler's and compiler's views of the size of an array is the same.

```
#include <cassert.h>

define MY_STRUCT_SIZE 8 /* Used by assembler source files */

struct my_struct {
    uint32_t arg1;
    uint32_t arg2;
};

CASSERT(MY_STRUCT_SIZE == sizeof(struct my_struct), assert_my_struct_size_mismatch);
```

If `MY_STRUCT_SIZE` in the above example were wrong then the compiler would emit an error like this:

```
my_struct.h:10:1: error: size of array 'assert_my_struct_size_mismatch' is negative
```

Using assert() to check for programming errors

In general, each secure world TF image (BL1, BL2, BL31 and BL32) should be treated as a tightly integrated package; the image builder should be aware of and responsible for all functionality within the image, even if code within that image is provided by multiple entities. This allows us to be more aggressive in interpreting invalid state or bad function arguments as programming errors using `assert()`, including arguments passed across platform porting interfaces. This is in contrast to code in a Linux environment, which is less tightly integrated and may attempt to be more defensive by passing the error back up the call stack.

Where possible, badly written TF code should fail early using `assert()`. This helps reduce the amount of untested conditional code. By default these statements are not compiled into release builds, although this can be overridden using the `ENABLE_ASSERTIONS` build flag.

Examples:

- Bad argument supplied to library function
- Bad argument provided by platform porting function
- Internal secure world image state is inconsistent

Handling integration errors

Each secure world image may be provided by a different entity (for example, a Trusted Boot vendor may provide the BL2 image, a TEE vendor may provide the BL32 image and the OEM/SoC vendor may provide the other images).

An image may contain bugs that are only visible when the images are integrated. The system integrator may not even have access to the debug variants of all the images in order to check if asserts are firing. For example, the release variant of BL1 may have already been burnt into the SoC. Therefore, TF code that detects an integration error should _not_ consider this a programming error, and should always take action, even in release builds.

If an integration error is considered non-critical it should be treated as a recoverable error. If the error is considered critical it should be treated as an unexpected unrecoverable error.

Handling recoverable errors

The secure world **must not** crash when supplied with bad data from an external source. For example, data from the normal world or a hardware device. Similarly, the secure world **must not** crash if it detects a non-critical problem within itself or the system. It must make every effort to recover from the problem by emitting a `WARN` message, performing any necessary error handling and continuing.

Examples:

- Secure world receives SMC from normal world with bad arguments.
- Secure world receives SMC from normal world at an unexpected time.
- BL31 receives SMC from BL32 with bad arguments.
- BL31 receives SMC from BL32 at unexpected time.
- Secure world receives recoverable error from hardware device. Retrying the operation may help here.
- Non-critical secure world service is not functioning correctly.
- BL31 SPD discovers minor configuration problem with corresponding SP.

Handling unrecoverable errors

In some cases it may not be possible for the secure world to recover from an error. This situation should be handled in one of the following ways:

1. If the unrecoverable error is unexpected then emit an `ERROR` message and call `panic()`. This will end up calling the platform-specific function `plat_panic_handler()`.
2. If the unrecoverable error is expected to occur in certain circumstances, then emit an `ERROR` message and call the platform-specific function `plat_error_handler()`.

Cases 1 and 2 are subtly different. A platform may implement `plat_panic_handler` and `plat_error_handler` in the same way (for example, by waiting for a secure watchdog to time-out or by invoking an interface on the platform's power controller to reset the platform). However, `plat_error_handler` may take additional action for some errors (for example, it may set a flag so the platform resets into a different mode). Also, `plat_panic_handler()` may implement additional debug functionality (for example, invoking a hardware breakpoint).

Examples of unexpected unrecoverable errors:

- BL32 receives an unexpected SMC response from BL31 that it is unable to recover from.
- BL31 Trusted OS SPD code discovers that BL2 has not loaded the corresponding Trusted OS, which is critical for platform operation.
- Secure world discovers that a critical hardware device is an unexpected and unrecoverable state.
- Secure world receives an unexpected and unrecoverable error from a critical hardware device.
- Secure world discovers that it is running on unsupported hardware.

Examples of expected unrecoverable errors:

- BL1/BL2 fails to load the next image due to missing/corrupt firmware on disk.
- BL1/BL2 fails to authenticate the next image due to an invalid certificate.
- Secure world continuously receives recoverable errors from a hardware device but is unable to proceed without a valid response.

Handling critical unresponsiveness

If the secure world is waiting for a response from an external source (for example, the normal world or a hardware device) which is critical for continued operation, it must not wait indefinitely. It must have a mechanism (for example, a secure watchdog) for resetting itself and/or the external source to prevent the system from executing in this state indefinitely.

Examples:

- BL1 is waiting for the normal world to raise an SMC to proceed to the next stage of the secure firmware update process.
- A Trusted OS is waiting for a response from a proxy in the normal world that is critical for continued operation.
- Secure world is waiting for a hardware response that is critical for continued operation.

3.4.6 Use of built-in *C* and *libc* data types

The *TF-A* codebase should be kept as portable as possible, especially since both 64-bit and 32-bit platforms are supported. To help with this, the following data type usage guidelines should be followed:

- Where possible, use the built-in *C* data types for variable storage (for example, `char`, `int`, `long long`, etc) instead of the standard *C99* types. Most code is typically only concerned with the minimum size of the data stored, which the built-in *C* types guarantee.
- Avoid using the exact-size standard *C99* types in general (for example, `uint16_t`, `uint32_t`, `uint64_t`, etc) since they can prevent the compiler from making optimizations. There are legitimate uses for them, for example to represent data of a known structure. When using them in struct definitions, consider how padding in the struct will work across architectures. For example, extra padding may be introduced in *AArch32* systems if a struct member crosses a 32-bit boundary.
- Use `int` as the default integer type - it's likely to be the fastest on all systems. Also this can be assumed to be 32-bit as a consequence of the Procedure Call Standard for the Arm Architecture and the Procedure Call Standard for the Arm 64-bit Architecture .
- Avoid use of `short` as this may end up being slower than `int` in some systems. If a variable must be exactly 16-bit, use `int16_t` or `uint16_t`.
- Avoid use of `long`. This is guaranteed to be at least 32-bit but, given that *int* is 32-bit on Arm platforms, there is no use for it. For integers of at least 64-bit, use `long long`.
- Use `char` for storing text. Use `uint8_t` for storing other 8-bit data.
- Use `unsigned` for integers that can never be negative (counts, indices, sizes, etc). TF intends to comply with MISRA "essential type" coding rules (10.X), where signed and unsigned types are considered different essential types. Choosing the correct type will aid this. MISRA static analysers will pick up any implicit signed/unsigned conversions that may lead to unexpected behaviour.
- For pointer types:
  - If an argument in a function declaration is pointing to a known type then simply use a pointer to that type (for example: `struct my_struct *`).
  - If a variable (including an argument in a function declaration) is pointing to a general, memory-mapped address, an array of pointers or another structure that is likely to require pointer arithmetic then use `uintptr_t`. This will reduce the amount of casting required in the code. Avoid using `unsigned long` or `unsigned long long` for this purpose; it may work but is less portable.
  - For other pointer arguments in a function declaration, use `void *`. This includes pointers to types that are abstracted away from the known API and pointers to arbitrary data. This allows the calling function to pass a pointer argument to the function without any explicit casting (the cast to `void *` is implicit). The function implementation can then do the appropriate casting to a specific type.
  - Avoid pointer arithmetic generally (as this violates MISRA C 2012 rule 18.4) and especially on void pointers (as this is only supported via language extensions and is considered non-standard). In TF-A, setting the `W` build flag to `W=3` enables the *-Wpointer-arith* compiler flag and this will emit warnings where pointer arithmetic is used.
  - Use `ptrdiff_t` to compare the difference between 2 pointers.
- Use `size_t` when storing the `sizeof()` something.
- Use `ssize_t` when returning the `sizeof()` something from a function that can also return an error code; the signed type allows for a negative return code in case of error. This practice should be used sparingly.
- Use `u_register_t` when it's important to store the contents of a register in its native size (32-bit in *AArch32* and 64-bit in *AArch64*). This is not a standard *C99* type but is widely available in libc implementations, including the FreeBSD version included with the TF codebase. Where possible, cast the variable to a more appropriate

type before interpreting the data. For example, the following struct in `ep_info.h` could use this type to minimize the storage required for the set of registers:

```
typedef struct aapcs64_params {
        u_register_t arg0;
        u_register_t arg1;
        u_register_t arg2;
        u_register_t arg3;
        u_register_t arg4;
        u_register_t arg5;
        u_register_t arg6;
        u_register_t arg7;
} aapcs64_params_t;
```

If some code wants to operate on `arg0` and knows that it represents a 32-bit unsigned integer on all systems, cast it to `unsigned int`.

These guidelines should be updated if additional types are needed.

3.4.7 Favor C language over assembly language

Generally, prefer code written in C over assembly. Assembly code is less portable, harder to understand, maintain and audit security wise. Also, static analysis tools generally don't analyze assembly code.

There are, however, legitimate uses of assembly language. These include:

- Early boot code executed before the C runtime environment is setup.
- Exception handling code.
- Low-level code where the exact sequence of instructions executed on the CPU matters, such as CPU reset sequences.
- Low-level code where specific system-level instructions must be used, such as cache maintenance operations.

3.5 Contributor's Guide

3.5.1 Getting Started

- Make sure you have a Github account and you are logged on both developer.trustedfirmware.org and review.trustedfirmware.org.
- If you plan to contribute a major piece of work, it is usually a good idea to start a discussion around it on the mailing list. This gives everyone visibility of what is coming up, you might learn that somebody else is already working on something similar or the community might be able to provide some early input to help shaping the design of the feature.

  If you intend to include Third Party IP in your contribution, please mention it explicitly in the email thread and ensure that the changes that include Third Party IP are made in a separate patch (or patch series).
- Clone Trusted Firmware-A on your own machine as described in *Getting the TF-A Source*.
- Create a local topic branch based on the Trusted Firmware-A `master` branch.

3.5.2 Making Changes

- Make commits of logical units. See these general Git guidelines for contributing to a project.
- Keep the commits on topic. If you need to fix another bug or make another enhancement, please address it on a separate topic branch.
- Split the patch in manageable units. Small patches are usually easier to review so this will speed up the review process.
- Avoid long commit series. If you do have a long series, consider whether some commits should be squashed together or addressed in a separate topic.
- Ensure that each commit in the series has at least one `Signed-off-by:` line, using your real name and email address. The names in the `Signed-off-by:` and `Commit:` lines must match. By adding this line the contributor certifies the contribution is made under the terms of the `Developer Certificate of Origin`.

  There might be multiple `Signed-off-by:` lines, depending on the history of the patch.

  More details may be found in the Gerrit Signed-off-by Lines guidelines.
- Ensure that each commit also has a unique `Change-Id:` line. If you have cloned the repository with the "*Clone with commit-msg hook*" clone method (following the *Prerequisites* document), this should already be the case.

  More details may be found in the Gerrit Change-Ids documentation.
- Write informative and comprehensive commit messages. A good commit message provides all the background information needed for reviewers to understand the intent and rationale of the patch. This information is also useful for future reference.

  For example:

  - What does the patch do?
  - What motivated it?
  - What impact does it have?
  - How was it tested?
  - Have alternatives been considered? Why did you choose this approach over another one?
  - If it fixes an issue, include a reference.
- Follow the *Coding Style* and *Coding Guidelines*.
  - Use the checkpatch.pl script provided with the Linux source tree. A Makefile target is provided for convenience, see *this section* for more details.
- Where appropriate, please update the documentation.
  - Consider whether the *Porting Guide*, *Firmware Design* document or other in-source documentation needs updating.
  - If you are submitting new files that you intend to be the code owner for (for example, a new platform port), then also update the *Code owners* file.
  - For topics with multiple commits, you should make all documentation changes (and nothing else) in the last commit of the series. Otherwise, include the documentation changes within the single commit.
- 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.

For significant contributions you may add your own copyright notice in the following format:

```
Portions copyright (c) [XXXX-]YYYY, <OWNER>. All rights reserved.
```

where XXXX is the year of first contribution (if different to YYYY) and YYYY is the year of most recent contribution. <OWNER> is your name or your company name.

- Ensure that each patch in the patch series compiles in all supported configurations. Patches which do not compile will not be merged.
- Please test your changes. As a minimum, ensure that Linux boots on the Foundation FVP. See *Arm Fixed Virtual Platforms (FVP)* for more information. For more extensive testing, consider running the TF-A Tests against your patches.
- Ensure that all CI automated tests pass. Failures should be fixed. They might block a patch, depending on how critical they are.

3.5.3 Submitting Changes

- Submit your changes for review at https://review.trustedfirmware.org targeting the `integration` branch.
- Add reviewers for your patch:
  - At least one code owner for each module modified by the patch. See the list of modules and their *Code owners*.
  - At least one maintainer. See the list of *Maintainers*.
  - If some module has no code owner, try to identify a suitable (non-code owner) reviewer. Running `git blame` on the module's source code can help, as it shows who has been working the most recently on this area of the code.

    Alternatively, if it is impractical to identify such a reviewer, you might send an email to the TF-A mailing list to broadcast your review request to the community.

  Note that self-reviewing a patch is prohibited, even if the patch author is the only code owner of a module modified by the patch. Getting a second pair of eyes on the code is essential to keep up with the quality standards the project aspires to.
- The changes will then undergo further review by the designated people. Any review comments will be made directly on your patch. This may require you to do some rework. For controversial changes, the discussion might be moved to the TF-A mailing list to involve more of the community.

  Refer to the Gerrit Uploading Changes documentation for more details.
- The patch submission rules are the following. For a patch to be approved and merged in the tree, it must get:
  - One `Code-Owner-Review+1` for each of the modules modified by the patch.
  - A `Maintainer-Review+1`.

  In the case where a code owner could not be found for a given module, `Code-Owner-Review+1` is substituted by `Code-Review+1`.

  In addition to these various code review labels, the patch must also get a `Verified+1`. This is usually set by the Continuous Integration (CI) bot when all automated tests passed on the patch. Sometimes, some of these automated tests may fail for reasons unrelated to the patch. In this case, the maintainers might (after analysis of the failures) override the CI bot score to certify that the patch has been correctly tested.

In the event where the CI system lacks proper tests for a patch, the patch author or a reviewer might agree to perform additional manual tests in their review and the reviewer incorporates the review of the additional testing in the `Code-Review+1` or `Code-Owner-Review+1` as applicable to attest that the patch works as expected. Where possible additional tests should be added to the CI system as a follow up task. For example, for a platform-dependent patch where the said platform is not available in the CI system's board farm.

- When the changes are accepted, the *Maintainers* will integrate them.
  - Typically, the *Maintainers* will merge the changes into the `integration` branch.
  - If the changes are not based on a sufficiently-recent commit, or if they cannot be automatically rebased, then the *Maintainers* may rebase it on the `integration` branch or ask you to do so.
  - After final integration testing, the changes will make their way into the `master` branch. If a problem is found during integration, the *Maintainers* will request your help to solve the issue. They may revert your patches and ask you to resubmit a reworked version of them or they may ask you to provide a fix-up patch.

3.5.4 Binary Components

- Platforms may depend on binary components submitted to the Trusted Firmware binary repository if they require code that the contributor is unable or unwilling to open-source. This should be used as a rare exception.
- All binary components must follow the contribution guidelines (in particular licensing rules) outlined in the readme.rst file of the binary repository.
- Binary components must be restricted to only the specific functionality that cannot be open-sourced and must be linked into a larger open-source platform port. The majority of the platform port must still be implemented in open source. Platform ports that are merely a thin wrapper around a binary component that contains all the actual code will not be accepted.
- Only platform port code (i.e. in the `plat/<vendor>` directory) may rely on binary components. Generic code must always be fully open-source.

3.6 Code Review Guidelines

This document provides TF-A specific details about the project's code review process. It should be read in conjunction with the Project Maintenance Process, which it supplements.

3.6.1 Why do we do code reviews?

The main goal of code reviews is to improve the code quality. By reviewing each other's code, we can help catch issues that were missed by the author before they are integrated in the source tree. Different people bring different perspectives, depending on their past work, experiences and their current use cases of TF-A in their products.

Code reviews also play a key role in sharing knowledge within the community. People with more expertise in one area of the code base can help those that are less familiar with it.

Code reviews are meant to benefit everyone through team work. It is not about unfairly criticizing or belittling the work of any contributor.

3.6.2 Good practices

To ensure the code review gives the greatest possible benefit, participants in the project should:

- Be considerate of other people and their needs. Participants may be working to different timescales, and have different priorities. Keep this in mind - be gracious while waiting for action from others, and timely in your actions when others are waiting for you.
- Review other people's patches where possible. The more active reviewers there are, the more quickly new patches can be reviewed and merged. Contributing to code review helps everyone in the long run, as it creates a culture of participation which serves everyone's interests.

3.6.3 Guidelines for patch contributors

In addition to the rules outlined in the *Contributor's Guide*, as a patch contributor you are expected to:

- Answer all comments from people who took the time to review your patches.
- Be patient and resilient. It is quite common for patches to go through several rounds of reviews and rework before they get approved, especially for larger features.

  In the event that a code review takes longer than you would hope for, you may try the following actions to speed it up:
- Ping the reviewers on Gerrit or on the mailing list. If it is urgent, explain why. Please remain courteous and do not abuse this.
- If one code owner has become unresponsive, ask the other code owners for help progressing the patch.
- If there is only one code owner and they have become unresponsive, ask one of the project maintainers for help.
- Do the right thing for the project, not the fastest thing to get code merged.

  For example, if some existing piece of code - say a driver - does not quite meet your exact needs, go the extra mile and extend the code with the missing functionality you require - as opposed to copying the code into some other directory to have the freedom to change it in any way. This way, your changes benefit everyone and will be maintained over time.

3.6.4 Guidelines for all reviewers

There are no good or bad review comments. If you have any doubt about a patch or need some clarifications, it's better to ask rather than letting a potential issue slip. Examples of review comments could be:

- Questions ("Why do you need to do this?", "What if X happens?")
- Bugs ("I think you need a logical || rather than a bitwise |.")
- Design issues ("This won't scale well when we introduce feature X.")
- Improvements ("Would it be better if we did Y instead?")

3.6.5 Guidelines for code owners

Code owners are listed on the *Project Maintenance* page, along with the module(s) they look after.

When reviewing a patch, code owners are expected to check the following:

- The patch looks good from a technical point of view. For example:
- The structure of the code is clear.
- It complies with the relevant standards or technical documentation (where applicable).
- It leverages existing interfaces rather than introducing new ones unnecessarily.
- It fits well in the design of the module.
- It adheres to the security model of the project. In particular, it does not increase the attack surface (e.g. new SMCs) without justification.
- The patch adheres to the TF-A *Coding Style*. The CI system should help catch coding style violations.
- (Only applicable to generic code) The code is MISRA-compliant (see *MISRA Compliance*). The CI system should help catch violations.
- Documentation is provided/updated (where applicable).
- The patch has had an appropriate level of testing. Testing details are expected to be provided by the patch author. If they are not, do not hesitate to request this information.
- All CI automated tests pass.

If a code owner is happy with a patch, they should give their approval through the `Code-Owner-Review+1` label in Gerrit. If instead, they have concerns, questions, or any other type of blocking comment, they should set `Code-Owner-Review-1`.

Code owners are expected to behave professionally and responsibly. Here are some guidelines for them:

- Once you are engaged in a review, make sure you stay involved until the patch is merged. Rejecting a patch and going away is not very helpful. You are expected to monitor the patch author's answers to your review comments, answer back if needed and review new revisions of their patch.
- Provide constructive feedback. Just saying, "This is wrong, you should do X instead." is usually not very helpful. The patch author is unlikely to understand why you are requesting this change and might feel personally attacked.
- Be mindful when reviewing a patch. As a code owner, you are viewed as the expert for the relevant module. By approving a patch, you are partially responsible for its quality and the effects it has for all TF-A users. Make sure you fully understand what the implications of a patch might be.

3.6.6 Guidelines for maintainers

Maintainers are listed on the *Project Maintenance* page.

When reviewing a patch, maintainers are expected to check the following:

- The general structure of the patch looks good. This covers things like:
  - Code organization.
  - Files and directories, names and locations.

    For example, platform code should be added under the `plat/` directory.

    - Naming conventions.

      For example, platform identifiers should be properly namespaced to avoid name clashes with generic code.

    - API design.

- Interaction of the patch with other modules in the code base.
- The patch aims at complying with any standard or technical documentation that applies.
- New files must have the correct license and copyright headers. See *this paragraph* for more information. The CI system should help catch files with incorrect or no copyright/license headers.
- There is no third party code or binary blobs with potential IP concerns. Maintainers should look for copyright or license notices in code, and use their best judgement. If they are unsure about a patch, they should ask other maintainers for help.
- Generally speaking, new driver code should be placed in the generic layer. There are cases where a driver has to stay into the platform layer but this should be the exception, rather than the rule.
- Existing common drivers (in particular for Arm IPs like the GIC driver) should not be copied into the platform layer to cater for platform quirks. This type of code duplication hurts the maintainability of the project. The duplicate driver is less likely to benefit from bug fixes and future enhancements. In most cases, it is possible to rework a generic driver to make it more flexible and fit slightly different use cases. That way, these enhancements benefit everyone.
- When a platform specific driver really is required, the burden lies with the patch author to prove the need for it. A detailed justification should be posted via the commit message or on the mailing list.
- Before merging a patch, verify that all review comments have been addressed. If this is not the case, encourage the patch author and the relevant reviewers to resolve these together.

If a maintainer is happy with a patch, they should give their approval through the `Maintainer-Review+1` label in Gerrit. If instead, they have concerns, questions, or any other type of blocking comment, they should set `Maintainer-Review-1`.

3.7 Frequently-Asked Questions (FAQ)

3.7.1 How do I update my changes?

Often it is necessary to update your patch set before it is merged. Refer to the Gerrit Upload Patch Set documentation on how to do so.

If you need to modify an existing patch set with multiple commits, refer to the Gerrit Replace Changes documentation.

3.7.2 How long will my changes take to merge into `integration`?

This can vary a lot, depending on:

- How important the patch set is considered by the TF maintainers. Where possible, you should indicate the required timescales for merging the patch set and the impact of any delay. Feel free to add a comment to your patch set to get an estimate of when it will be merged.
- The quality of the patch set. Patches are likely to be merged more quickly if they follow the coding guidelines, have already had some code review, and have been appropriately tested.
- The impact of the patch set. For example, a patch that changes a key generic API is likely to receive much greater scrutiny than a local change to a specific platform port.
- How much opportunity for external review is required. For example, the TF maintainers may not wait for external review comments to merge trivial bug-fixes but may wait up to a week to merge major changes, or ones requiring feedback from specific parties.
- How many other patch sets are waiting to be integrated and the risk of conflict between the topics.
- If there is a code freeze in place in preparation for the release. Please refer the *Release Processes* document for more details.
- The workload of the TF maintainers.

3.7.3 How long will it take for my changes to go from `integration` to `master`?

This depends on how many concurrent patches are being processed at the same time. In simple cases where all potential regressions have already been tested, the delay will be less than 1 day. If the TF maintainers are trying to merge several things over the course of a few days, it might take up to a week. Typically, it will be 1-2 days.

The worst case is if the TF maintainers are trying to make a release while also receiving patches that will not be merged into the release. In this case, the patches will be merged onto `integration`, which will temporarily diverge from the release branch. The `integration` branch will be rebased onto `master` after the release, and then `master` will be fast-forwarded to `integration` 1-2 days later. This whole process could take up 4 weeks. Please refer to the *Release Processes* document for code freeze dates. The TF maintainers will inform the patch owner if this is going to happen.

It is OK to create a patch based on commits that are only available in `integration` or another patch set, rather than `master`. There is a risk that the dependency commits will change (for example due to patch set rework or integration problems). If this happens, the dependent patch will need reworking.

3.7.4 What are these strange comments in my changes?

All the comments from `ci-bot-user` are associated with Continuous Integration infrastructure. The links published on the comment are not currently accessible, but would be after the CI has been transitioned to trustedfirmware.org.

3.8 Secure Development Guidelines

This page contains guidance on what to check for additional security measures, including build options that can be modified to improve security or catch issues early in development.

3.8.1 Security considerations

Part of the security of a platform is handling errors correctly, as described in the previous section. There are several other security considerations covered in this section.

Do not leak secrets to the normal world

The secure world **must not** leak secrets to the normal world, for example in response to an SMC.

Handling Denial of Service attacks

The secure world **should never** crash or become unusable due to receiving too many normal world requests (a *Denial of Service* or *DoS* attack). It should have a mechanism for throttling or ignoring normal world requests.

Preventing Secure-world timing information leakage via PMU counters

The Secure world needs to implement some defenses to prevent the Non-secure world from making it leak timing information. In general, higher privilege levels must defend from those below when the PMU is treated as an attack vector.

Refer to the *Performance Monitoring Unit* guide for detailed information on the PMU registers.

Timing leakage attacks from the Non-secure world

Since the Non-secure world has access to the `PMCR` register, it can configure the PMU to increment counters at any exception level and in both Secure and Non-secure state. Thus, it attempts to leak timing information from the Secure world.

Shown below is an example of such a configuration:

- `PMEVTYPER0_EL0` and `PMCCFILTR_EL0`:
  - Set `P` to `0`.
  - Set `NSK` to `1`.
  - Set `M` to `0`.
  - Set `NSH` to `0`.
  - Set `SH` to `1`.
- `PMCNTENSET_EL0`:
  - Set `P[0]` to `1`.
  - Set `C` to `1`.
- `PMCR_EL0`:
  - Set `DP` to `0`.

    - Set E to 1.

This configuration instructs PMEVCNTR0_EL0 and PMCCNTR_EL0 to increment at Secure EL1, Secure EL2 (if implemented) and EL3.

Since the Non-secure world has fine-grained control over where (at which exception levels) it instructs counters to increment, obtaining event counts would allow it to carry out side-channel timing attacks against the Secure world. Examples include Spectre, Meltdown, as well as extracting secrets from cryptographic algorithms with data-dependent variations in their execution time.

Secure world mitigation strategies

The MDCR_EL3 register allows EL3 to configure the PMU (among other things). The Arm ARM details all of the bit fields in this register, but for the PMU there are two bits which determine the permissions of the counters:

- SPME for the programmable counters.
- SCCD for the cycle counter.

Depending on the implemented features, the Secure world can prohibit counting in AArch64 state via the following:

- ARMv8.2-Debug not implemented:
    - Prohibit general event counters and the cycle counter: `MDCR_EL3.SPME == 0 && PMCR_EL0.DP == 1 && !ExternalSecureNoninvasiveDebugEnabled()`.
        - MDCR_EL3.SPME resets to 0, so by default general events should not be counted in the Secure world.
        - The PMCR_EL0.DP bit therefore needs to be set to 1 when EL3 is entered and PMCR_EL0 needs to be saved and restored in EL3.
        - ExternalSecureNoninvasiveDebugEnabled() is an authentication interface which is implementation-defined unless ARMv8.4-Debug is implemented. The Arm ARM has detailed information on this topic.
    - The only other way is to disable the PMCR_EL0.E bit upon entering EL3, which disables counting altogether.
- ARMv8.2-Debug implemented:
    - Prohibit general event counters: `MDCR_EL3.SPME == 0`.
    - Prohibit cycle counter: `MDCR_EL3.SPME == 0 && PMCR_EL0.DP == 1`. PMCR_EL0 therefore needs to be saved and restored in EL3.
- ARMv8.5-PMU implemented:
    - Prohibit general event counters: as in ARMv8.2-Debug.
    - Prohibit cycle counter: `MDCR_EL3.SCCD == 1`

In Aarch32 execution state the MDCR_EL3 alias is the SDCR register, which has some of the bit fields of MDCR_EL3, most importantly the SPME and SCCD bits.

3.8.2 Build options

Several build options can be used to check for security issues. Refer to the *Build Options* for detailed information on these.

- The `BRANCH_PROTECTION` build flag can be used to enable Pointer Authentication and Branch Target Identification.
- The `ENABLE_STACK_PROTECTOR` build flag can be used to identify buffer overflows.
- The `W` build flag can be used to enable a number of compiler warning options to detect potentially incorrect code.
  - W=0 (default value)

    The `Wunused` with `Wno-unused-parameter`, `Wdisabled-optimization` and `Wvla` flags are enabled.

    The `Wunused-but-set-variable`, `Wmaybe-uninitialized` and `Wpacked-bitfield-compat` are GCC specific flags that are also enabled.
  - W=1

    Adds `Wextra`, `Wmissing-format-attribute`, `Wmissing-prototypes`, `Wold-style-definition` and `Wunused-const-variable`.
  - W=2

    Adds `Waggregate-return`, `Wcast-align`, `Wnested-externs`, `Wshadow`, `Wlogical-op`.
  - W=3

    Adds `Wbad-function-cast`, `Wcast-qual`, `Wconversion`, `Wpacked`, `Wpointer-arith`, `Wredundant-decls` and `Wswitch-default`.

  Refer to the GCC or Clang documentation for more information on the individual options: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html and https://clang.llvm.org/docs/DiagnosticsReference.html.

  NB: The `Werror` flag is enabled by default in TF-A and can be disabled by setting the `E` build flag to 0.

References

- Arm ARM

CHAPTER

FOUR

COMPONENTS

4.1 Secure Payload Dispatcher (SPD)

4.1.1 OP-TEE Dispatcher

OP-TEE OS is a Trusted OS running as Secure EL1.

To build and execute OP-TEE follow the instructions at OP-TEE build.git

---

4.1.2 Trusted Little Kernel (TLK) Dispatcher

TLK dispatcher (TLK-D) adds support for NVIDIA's Trusted Little Kernel (TLK) to work with Trusted Firmware-A (TF-A). TLK-D can be compiled by including it in the platform's makefile. TLK is primarily meant to work with Tegra SoCs, so while TF-A only supports TLK on Tegra, the dispatcher code can only be compiled for other platforms.

In order to compile TLK-D, we need a BL32 image to be present. Since, TLKD just needs to compile, any BL32 image would do. To use TLK as the BL32, please refer to the "Build TLK" section.

Once a BL32 is ready, TLKD can be included in the image by adding "SPD=tlkd" to the build command.

Trusted Little Kernel (TLK)

TLK is a Trusted OS running as Secure EL1. It is a Free Open Source Software (FOSS) release of the NVIDIA® Trusted Little Kernel (TLK) technology, which extends technology made available with the development of the Little Kernel (LK). You can download the LK modular embedded preemptive kernel for use on Arm, x86, and AVR32 systems from https://github.com/travisg/lk

NVIDIA implemented its Trusted Little Kernel (TLK) technology, designed as a free and open-source trusted execution environment (OTE).

TLK features include:

- Small, pre-emptive kernel
- Supports multi-threading, IPCs, and thread scheduling
- Added TrustZone features
- Added Secure Storage
- Under MIT/FreeBSD license

NVIDIA extensions to Little Kernel (LK) include:

- User mode
- Address-space separation for TAs
- TLK Client Application (CA) library
- TLK TA library
- Crypto library (encrypt/decrypt, key handling) via OpenSSL
- Linux kernel driver
- Cortex A9/A15 support
- Power Management
- TrustZone memory carve-out (reconfigurable)
- Page table management
- Debugging support over UART (USB planned)

TLK is hosted by NVIDIA on http://nv-tegra.nvidia.com under the 3rdparty/ote_partner/tlk.git repository. Detailed information about TLK and OTE can be found in the Tegra_BSP_for_Android_TLK_FOSS_Reference.pdf manual located under the "documentation" directory_.

Build TLK

To build and execute TLK, follow the instructions from "Building a TLK Device" section from Tegra_BSP_for_Android_TLK_FOSS_Reference.pdf manual.

Input parameters to TLK

TLK expects the TZDRAM size and a structure containing the boot arguments. BL2 passes this information to the EL3 software as members of the bl32_ep_info struct, where bl32_ep_info is part of bl31_params_t (passed by BL2 in X0)

Example

```
bl32_ep_info->args.arg0 = TZDRAM size available for BL32
bl32_ep_info->args.arg1 = unused (used only on Armv7-A)
bl32_ep_info->args.arg2 = pointer to boot args
```

4.1.3 Trusty Dispatcher

Trusty is a a set of software components, supporting a Trusted Execution Environment (TEE) on mobile devices, published and maintained by Google.

Detailed information and build instructions can be found on the Android Open Source Project (AOSP) webpage for Trusty hosted at https://source.android.com/security/trusty

Boot parameters

Custom boot parameters can be passed to Trusty by providing a platform specific function:

```
void plat_trusty_set_boot_args(aapcs64_params_t *args)
```

If this function is provided `args->arg0` must be set to the memory size allocated to trusty. If the platform does not provide this function, but defines `TSP_SEC_MEM_SIZE`, a default implementation will pass the memory size from `TSP_SEC_MEM_SIZE`. `args->arg1` can be set to a platform specific parameter block, and `args->arg2` should then be set to the size of that block.

Supported platforms

Out of all the platforms supported by Trusted Firmware-A, Trusty is only verified and supported by NVIDIA's Tegra SoCs.

4.2 Arm SiP Services

This document enumerates and describes the Arm SiP (Silicon Provider) services.

SiP services are non-standard, platform-specific services offered by the silicon implementer or platform provider. They are accessed via `SMC` ("SMC calls") instruction executed from Exception Levels below EL3. SMC calls for SiP services:

- Follow SMC Calling Convention;
- Use SMC function IDs that fall in the SiP range, which are `0xc2000000` - `0xc200ffff` for 64-bit calls, and `0x82000000` - `0x8200ffff` for 32-bit calls.

The Arm SiP implementation offers the following services:

- Performance Measurement Framework (PMF)
- Execution State Switching service
- DebugFS interface

Source definitions for Arm SiP service are located in the `arm_sip_svc.h` header file.

4.2.1 Performance Measurement Framework (PMF)

The *Performance Measurement Framework* allows callers to retrieve timestamps captured at various paths in TF-A execution.

4.2.2 Execution State Switching service

Execution State Switching service provides a mechanism for a non-secure lower Exception Level (either EL2, or NS EL1 if EL2 isn't implemented) to request to switch its execution state (a.k.a. Register Width), either from AArch64 to AArch32, or from AArch32 to AArch64, for the calling CPU. This service is only available when Trusted Firmware-A (TF-A) is built for AArch64 (i.e. when build option `ARCH` is set to `aarch64`).

ARM_SIP_SVC_EXE_STATE_SWITCH

```
Arguments:
    uint32_t Function ID
    uint32_t PC hi
    uint32_t PC lo
    uint32_t Cookie hi
    uint32_t Cookie lo

Return:
    uint32_t
```

The function ID parameter must be `0x82000020`. It uniquely identifies the Execution State Switching service being requested.

The parameters *PC hi* and *PC lo* defines upper and lower words, respectively, of the entry point (physical address) at which execution should start, after Execution State has been switched. When calling from AArch64, *PC hi* must be 0.

When execution starts at the supplied entry point after Execution State has been switched, the parameters *Cookie hi* and *Cookie lo* are passed in CPU registers 0 and 1, respectively. When calling from AArch64, *Cookie hi* must be 0.

This call can only be made on the primary CPU, before any secondaries were brought up with `CPU_ON` PSCI call. Otherwise, the call will always fail.

The effect of switching execution state is as if the Exception Level were entered for the first time, following power on. This means CPU registers that have a defined reset value by the Architecture will assume that value. Other registers should not be expected to hold their values before the call was made. CPU endianness, however, is preserved from the previous execution state. Note that this switches the execution state of the calling CPU only. This is not a substitute for PSCI `SYSTEM_RESET`.

The service may return the following error codes:

- `STATE_SW_E_PARAM`: If any of the parameters were deemed invalid for a specific request.
- `STATE_SW_E_DENIED`: If the call is not successful, or when TF-A is built for AArch32.

If the call is successful, the caller wouldn't observe the SMC returning. Instead, execution starts at the supplied entry point, with the CPU registers 0 and 1 populated with the supplied *Cookie hi* and *Cookie lo* values, respectively.

4.2.3 DebugFS interface

The optional DebugFS interface is accessed through an SMC SiP service. Refer to the component documentation for details.

String parameters are passed through a shared buffer using a specific union:

```
union debugfs_parms {
    struct {
        char fname[MAX_PATH_LEN];
    } open;

    struct mount {
        char srv[MAX_PATH_LEN];
        char where[MAX_PATH_LEN];
        char spec[MAX_PATH_LEN];
    } mount;

    struct {
```

(continues on next page)

(continued from previous page)

```
        char path[MAX_PATH_LEN];
        dir_t dir;
    } stat;

    struct {
        char oldpath[MAX_PATH_LEN];
        char newpath[MAX_PATH_LEN];
    } bind;
};
```

Format of the dir_t structure as such:

```
typedef struct {
    char            name[NAMELEN];
    long            length;
    unsigned char   mode;
    unsigned char   index;
    unsigned char   dev;
    qid_t           qid;
} dir_t;
```

- Identifiers

SMC_OK	0
SMC_UNK	-1
DEBUGFS_E_INVALID_PARAMS	-2

MOUNT	0
CREATE	1
OPEN	2
CLOSE	3
READ	4
WRITE	5
SEEK	6
BIND	7
STAT	8
INIT	10
VERSION	11

MOUNT

Description

This operation mounts a blob of data pointed to by path stored in *src*, at filesystem location pointed to by path stored in *where*, using driver pointed to by path in *spec*.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`MOUNT`

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if mount operation failed

OPEN

Description

This operation opens the file path pointed to by *fname*.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`OPEN`
uint32_t	mode

mode can be one of:

```
enum mode {
    O_READ   = 1 << 0,
    O_WRITE  = 1 << 1,
    O_RDWR   = 1 << 2,
    O_BIND   = 1 << 3,
    O_DIR    = 1 << 4,
    O_STAT   = 1 << 5
};
```

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if open operation failed
uint32_t	w1: file descriptor id on success.

CLOSE

Description

This operation closes a file described by a file descriptor obtained by a previous call to OPEN.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`CLOSE`
uint32_t	File descriptor id returned by OPEN

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if close operation failed

READ

Description

This operation reads a number of bytes from a file descriptor obtained by a previous call to OPEN.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`READ`
uint32_t	File descriptor id returned by OPEN
uint32_t	Number of bytes to read

Return values

On success, the read data is retrieved from the shared buffer after the operation.

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if read operation failed
uint32_t	w1: number of bytes read on success.

SEEK

Description

Move file pointer for file described by given *file descriptor* of given *offset* related to *whence*.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`SEEK`
uint32_t	File descriptor id returned by OPEN
sint32_t	offset in the file relative to whence
uint32_t	whence

whence can be one of:

KSEEK_SET	0
KSEEK_CUR	1
KSEEK_END	2

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if seek operation failed

BIND

Description

Create a link from *oldpath* to *newpath*.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`BIND`

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if bind operation failed

STAT

Description

Perform a stat operation on provided file *name* and returns the directory entry statistics into *dir*.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`STAT`

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if stat operation failed

INIT

Description

Initial call to setup the shared exchange buffer. Notice if successful once, subsequent calls fail after a first initialization. The caller maps the same page frame in its virtual space and uses this buffer to exchange string parameters with filesystem primitives.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`INIT`
uint64_t	Physical address of the shared buffer.

Return values

int32_t	w0 == SMC_OK on success w0 == DEBUGFS_E_INVALID_PARAMS if already initialized, or internal error occurred.

VERSION

Description

Returns the debugfs interface version if implemented in TF-A.

Parameters

uint32_t	FunctionID (0x82000030 / 0xC2000030)
uint32_t	`VERSION`

Return values

int32_t	w0 == SMC_OK on success w0 == SMC_UNK if interface is not implemented
uint32_t	w1: On success, debugfs interface version, 32 bits value with major version number in upper 16 bits and minor version in lower 16 bits.

- CREATE(1) and WRITE (5) command identifiers are unimplemented and return *SMC_UNK*.

4.3 Debug FS

Contents

4.3.1 Overview

The *DebugFS* feature is primarily aimed at exposing firmware debug data to higher SW layers such as a non-secure component. Such component can be the TFTF test payload or a Linux kernel module.

4.3.2 Virtual filesystem

The core functionality lies in a virtual file system based on a 9p file server interface (Notes on the Plan 9 Kernel Source and Linux 9p remote filesystem protocol). The implementation permits exposing virtual files, firmware drivers, and file blobs.

Namespace

Two namespaces are exposed:

- # is used as root for drivers (e.g. #t0 is the first uart)
- / is used as root for virtual "files" (e.g. /fip, or /dev/uart)

9p interface

The associated primitives are:

- Unix-like:
  - open(): create a file descriptor that acts as a handle to the file passed as an argument.
  - close(): close the file descriptor created by open().
  - read(): read from a file to a buffer.
  - write(): write from a buffer to a file.
  - seek(): set the file position indicator of a file descriptor either to a relative or an absolute offset.
  - stat(): get information about a file (type, mode, size, …).

```
int open(const char *name, int flags);
int close(int fd);
int read(int fd, void *buf, int n);
int write(int fd, void *buf, int n);
int seek(int fd, long off, int whence);
int stat(char *path, dir_t *dir);
```

- Specific primitives :
  - mount(): create a link between a driver and spec.
  - create(): create a file in a specific location.
  - bind(): expose the content of a directory to another directory.

```
int mount(char *srv, char *mnt, char *spec);
int create(const char *name, int flags);
int bind(char *path, char *where);
```

This interface is embedded into the BL31 run-time payload when selected by build options. The interface multiplexes drivers or emulated "files":

- Debug data can be partitioned into different virtual files e.g. expose PMF measurements through a file, and internal firmware state counters through another file.
- This permits direct access to a firmware driver, mainly for test purposes (e.g. a hardware device that may not be accessible to non-privileged/ non-secure layers, or for which no support exists in the NS side).

4.3.3 SMC interface

The communication with the 9p layer in BL31 is made through an SMC conduit (SMC Calling Convention), using a specific SiP Function Id. An NS shared buffer is used to pass path string parameters, or e.g. to exchange data on a read operation. Refer to *ARM SiP Services* for a description of the SMC interface.

4.3.4 Security considerations

- Due to the nature of the exposed data, the feature is considered experimental and importantly **shall only be used in debug builds**.
- Several primitive imply string manipulations and usage of string formats.
- Special care is taken with the shared buffer to avoid TOCTOU attacks.

4.3.5 Limitations

- In order to setup the shared buffer, the component consuming the interface needs to allocate a physical page frame and transmit its address.
- In order to map the shared buffer, BL31 requires enabling the dynamic xlat table option.
- Data exchange is limited by the shared buffer length. A large read operation might be split into multiple read operations of smaller chunks.
- On concurrent access, a spinlock is implemented in the BL31 service to protect the internal work buffer, and re-entrancy into the filesystem layers.
- Notice, a physical device driver if exposed by the firmware may conflict with the higher level OS if the latter implements its own driver for the same physical device.

4.3.6 Applications

The SMC interface is accessible from an NS environment, that is:

- a test payload, bootloader or hypervisor running at NS-EL2
- a Linux kernel driver running at NS-EL1
- a Linux userspace application through the kernel driver

4.4 Exception Handling Framework

This document describes various aspects of handling exceptions by Runtime Firmware (BL31) that are targeted at EL3, other than SMCs. The *EHF* takes care of the following exceptions when targeted at EL3:

- Interrupts
- Synchronous External Aborts
- Asynchronous External Aborts

TF-A's handling of synchronous `SMC` exceptions raised from lower ELs is described in the *Firmware Design document*. However, the *EHF* changes the semantics of *Interrupt handling* and *synchronous exceptions* other than SMCs.

The *EHF* is selected by setting the build option `EL3_EXCEPTION_HANDLING` to `1`, and is only available for AArch64 systems.

4.4.1 Introduction

Through various control bits in the `SCR_EL3` register, the Arm architecture allows for asynchronous exceptions to be routed to EL3. As described in the *Interrupt Management Framework* document, depending on the chosen interrupt routing model, TF-A appropriately sets the `FIQ` and `IRQ` bits of `SCR_EL3` register to effect this routing. For most use cases, other than for the purpose of facilitating context switch between Normal and Secure worlds, FIQs and IRQs routed to EL3 are not required to be handled in EL3.

However, the evolving system and standards landscape demands that various exceptions are targeted at and handled in EL3. For instance:

- Starting with ARMv8.2 architecture extension, many RAS features have been introduced to the Arm architecture. With RAS features implemented, various components of the system may use one of the asynchronous exceptions to signal error conditions to PEs. These error conditions are of critical nature, and it's imperative that corrective or remedial actions are taken at the earliest opportunity. Therefore, a *Firmware-first Handling* approach is generally followed in response to RAS events in the system.
- The Arm SDEI specification defines interfaces through which Normal world interacts with the Runtime Firmware in order to request notification of system events. The *SDEI* specification requires that these events are notified even when the Normal world executes with the exceptions masked. This too implies that firmware-first handling is required, where the events are first received by the EL3 firmware, and then dispatched to Normal world through purely software mechanism.

For *TF-A*, firmware-first handling means that asynchronous exceptions are suitably routed to EL3, and the Runtime Firmware (BL31) is extended to include software components that are capable of handling those exceptions that target EL3. These components—referred to as *dispatchers*[1] in general—may choose to:

- Receive and handle exceptions entirely in EL3, meaning the exceptions handling terminates in EL3.
- Receive exceptions, but handle part of the exception in EL3, and delegate the rest of the handling to a dedicated software stack running at lower Secure ELs. In this scheme, the handling spans various secure ELs.
- Receive exceptions, but handle part of the exception in EL3, and delegate processing of the error to dedicated software stack running at lower secure ELs (as above); additionally, the Normal world may also be required to participate in the handling, or be notified of such events (for example, as an *SDEI* event). In this scheme, exception handling potentially and maximally spans all ELs in both Secure and Normal worlds.

On any given system, all of the above handling models may be employed independently depending on platform choice and the nature of the exception received.

[1] Not to be confused with *Secure Payload Dispatcher*, which is an EL3 component that operates in EL3 on behalf of Secure OS.

4.4.2 The role of Exception Handling Framework

Corollary to the use cases cited above, the primary role of the *EHF* is to facilitate firmware-first handling of exceptions on Arm systems. The *EHF* thus enables multiple exception dispatchers in runtime firmware to co-exist, register for, and handle exceptions targeted at EL3. This section outlines the basics, and the rest of this document expands the various aspects of the *EHF*.

In order to arbitrate exception handling among dispatchers, the *EHF* operation is based on a priority scheme. This priority scheme is closely tied to how the Arm GIC architecture defines it, although it's applied to non-interrupt exceptions too (SErrors, for example).

The platform is required to *partition* the Secure priority space into priority levels as applicable for the Secure software stack. It then assigns the dispatchers to one or more priority levels. The dispatchers then register handlers for the priority levels at runtime. A dispatcher can register handlers for more than one priority level.

A priority level is *active* when a handler at that priority level is currently executing in EL3, or has delegated the execution to a lower EL. For interrupts, this is implicit when an interrupt is targeted and acknowledged at EL3, and the priority of the acknowledged interrupt is used to match its registered handler. The priority level is likewise implicitly deactivated when the interrupt handling concludes by EOIing the interrupt.

Non-interrupt exceptions (SErrors, for example) don't have a notion of priority. In order for the priority arbitration to work, the *EHF* provides APIs in order for these non-interrupt exceptions to assume a priority, and to interwork with interrupts. Dispatchers handling such exceptions must therefore explicitly activate and deactivate the respective priority level as and when they're handled or delegated.

Because priority activation and deactivation for interrupt handling is implicit and involves GIC priority masking, it's impossible for a lower priority interrupt to preempt a higher priority one. By extension, this means that a lower priority dispatcher cannot preempt a higher-priority one. Priority activation and deactivation for non-interrupt exceptions, however, has to be explicit. The *EHF* therefore disallows for lower priority level to be activated whilst a higher priority level is active, and would result in a panic. Likewise, a panic would result if it's attempted to deactivate a lower priority level when a higher priority level is active.

In essence, priority level activation and deactivation conceptually works like a stack—priority levels stack up in strictly increasing fashion, and need to be unstacked in strictly the reverse order. For interrupts, the GIC ensures this is the case; for non-interrupts, the *EHF* monitors and asserts this. See *Transition of priority levels*.

4.4.3 Interrupt handling

The *EHF* is a client of *Interrupt Management Framework*, and registers the top-level handler for interrupts that target EL3, as described in the *Interrupt Management Framework* document. This has the following implications:

- On GICv3 systems, when executing in S-EL1, pending Non-secure interrupts of sufficient priority are signalled as FIQs, and therefore will be routed to EL3. As a result, S-EL1 software cannot expect to handle Non-secure interrupts at S-EL1. Essentially, this deprecates the routing mode described as *CSS=0, TEL3=0*.

  In order for S-EL1 software to handle Non-secure interrupts while having *EHF* enabled, the dispatcher must adopt a model where Non-secure interrupts are received at EL3, but are then *synchronously* handled over to S-EL1.

- On GICv2 systems, it's required that the build option `GICV2_G0_FOR_EL3` is set to `1` so that *Group 0* interrupts target EL3.
- While executing in Secure world, *EHF* sets GIC Priority Mask Register to the lowest Secure priority. This means that no Non-secure interrupts can preempt Secure execution. See *Effect on SMC calls* for more details.

As mentioned above, with *EHF*, the platform is required to partition *Group 0* interrupts into distinct priority levels. A dispatcher that chooses to receive interrupts can then *own* one or more priority levels, and register interrupt handlers for them. A given priority level can be assigned to only one handler. A dispatcher may register more than one priority level.

Dispatchers are assigned interrupt priority levels in two steps:

Partitioning priority levels

Interrupts are associated to dispatchers by way of grouping and assigning interrupts to a priority level. In other words, all interrupts that are to target a particular dispatcher should fall in a particular priority level. For priority assignment:

- Of the 8 bits of priority that Arm GIC architecture permits, bit 7 must be 0 (secure space).
- Depending on the number of dispatchers to support, the platform must choose to use the top *n* of the 7 remaining bits to identify and assign interrupts to individual dispatchers. Choosing *n* bits supports up to 2^n distinct dispatchers. For example, by choosing 2 additional bits (i.e., bits 6 and 5), the platform can partition into 4 secure priority ranges: `0x0`, `0x20`, `0x40`, and `0x60`. See *Interrupt handling example*.

Note: The Arm GIC architecture requires that a GIC implementation that supports two security states must implement at least 32 priority levels; i.e., at least 5 upper bits of the 8 bits are writeable. In the scheme described above, when choosing *n* bits for priority range assignment, the platform must ensure that at least `n+1` top bits of GIC priority are writeable.

The priority thus assigned to an interrupt is also used to determine the priority of delegated execution in lower ELs. Delegated execution in lower EL is associated with a priority level chosen with `ehf_activate_priority()` API (described *later*). The chosen priority level also determines the interrupts masked while executing in a lower EL, therefore controls preemption of delegated execution.

The platform expresses the chosen priority levels by declaring an array of priority level descriptors. Each entry in the array is of type `ehf_pri_desc_t`, and declares a priority level, and shall be populated by the `EHF_PRI_DESC()` macro.

Warning: The macro `EHF_PRI_DESC()` installs the descriptors in the array at a computed index, and not necessarily where the macro is placed in the array. The size of the array might therefore be larger than what it appears to be. The `ARRAY_SIZE()` macro therefore should be used to determine the size of array.

Finally, this array of descriptors is exposed to *EHF* via the `EHF_REGISTER_PRIORITIES()` macro.

Refer to the *Interrupt handling example* for usage. See also: *Interrupt Prioritisation Considerations*.

Programming priority

The text in *Partitioning priority levels* only describes how the platform expresses the required levels of priority. It however doesn't choose interrupts nor program the required priority in GIC.

The *Firmware Design guide* explains methods for configuring secure interrupts. *EHF* requires the platform to enumerate interrupt properties (as opposed to just numbers) of Secure interrupts. The priority of secure interrupts must match that as determined in the *Partitioning priority levels* section above.

See *Limitations*, and also refer to *Interrupt handling example* for illustration.

4.4.4 Registering handler

Dispatchers register handlers for their priority levels through the following API:

```
int ehf_register_priority_handler(int pri, ehf_handler_t handler)
```

The API takes two arguments:

- The priority level for which the handler is being registered;
- The handler to be registered. The handler must be aligned to 4 bytes.

If a dispatcher owns more than one priority levels, it has to call the API for each of them.

The API will succeed, and return `0`, only if:

- There exists a descriptor with the priority level requested.
- There are no handlers already registered by a previous call to the API.

Otherwise, the API returns `-1`.

The interrupt handler should have the following signature:

```
typedef int (*ehf_handler_t)(uint32_t intr_raw, uint32_t flags, void *handle,
                void *cookie);
```

The parameters are as obtained from the top-level *EL3 interrupt handler*.

The *SDEI dispatcher*, for example, expects the platform to allocate two different priority levels—`PLAT_SDEI_CRITICAL_PRI`, and `PLAT_SDEI_NORMAL_PRI` —and registers the same handler to handle both levels.

4.4.5 Interrupt handling example

The following annotated snippet demonstrates how a platform might choose to assign interrupts to fictitious dispatchers:

```
#include <common/interrupt_props.h>
#include <drivers/arm/gic_common.h>
#include <exception_mgmt.h>

...

/*
 * This platform uses 2 bits for interrupt association. In total, 3 upper
 * bits are in use.
 *
 *  7 6 5   3      0
 * .-.-.-.----------.
 * |0|b|b|  ..0..   |
 * '-'-'-'----------'
 */
#define PLAT_PRI_BITS        2

/* Priorities for individual dispatchers */
#define DISP0_PRIO           0x00 /* Not used */
#define DISP1_PRIO           0x20
#define DISP2_PRIO           0x40
```

(continues on next page)

(continued from previous page)

```
#define DISP3_PRIO          0x60

/* Install priority level descriptors for each dispatcher */
ehf_pri_desc_t plat_exceptions[] = {
     EHF_PRI_DESC(PLAT_PRI_BITS, DISP1_PRIO),
     EHF_PRI_DESC(PLAT_PRI_BITS, DISP2_PRIO),
     EHF_PRI_DESC(PLAT_PRI_BITS, DISP3_PRIO),
};

/* Expose priority descriptors to Exception Handling Framework */
EHF_REGISTER_PRIORITIES(plat_exceptions, ARRAY_SIZE(plat_exceptions),
     PLAT_PRI_BITS);

...

/* List interrupt properties for GIC driver. All interrupts target EL3 */
const interrupt_prop_t plat_interrupts[] = {
     /* Dispatcher 1 owns interrupts d1_0 and d1_1, so assigns priority DISP1_PRIO */
     INTR_PROP_DESC(d1_0, DISP1_PRIO, INTR_TYPE_EL3, GIC_INTR_CFG_LEVEL),
     INTR_PROP_DESC(d1_1, DISP1_PRIO, INTR_TYPE_EL3, GIC_INTR_CFG_LEVEL),

     /* Dispatcher 2 owns interrupts d2_0 and d2_1, so assigns priority DISP2_PRIO */
     INTR_PROP_DESC(d2_0, DISP2_PRIO, INTR_TYPE_EL3, GIC_INTR_CFG_LEVEL),
     INTR_PROP_DESC(d2_1, DISP2_PRIO, INTR_TYPE_EL3, GIC_INTR_CFG_LEVEL),

     /* Dispatcher 3 owns interrupts d3_0 and d3_1, so assigns priority DISP3_PRIO */
     INTR_PROP_DESC(d3_0, DISP3_PRIO, INTR_TYPE_EL3, GIC_INTR_CFG_LEVEL),
     INTR_PROP_DESC(d3_1, DISP3_PRIO, INTR_TYPE_EL3, GIC_INTR_CFG_LEVEL),
};

...

/* Dispatcher 1 registers its handler */
ehf_register_priority_handler(DISP1_PRIO, disp1_handler);

/* Dispatcher 2 registers its handler */
ehf_register_priority_handler(DISP2_PRIO, disp2_handler);

/* Dispatcher 3 registers its handler */
ehf_register_priority_handler(DISP3_PRIO, disp3_handler);

...
```

See also the *Build-time flow* and the *Run-time flow*.

4.4.6 Activating and Deactivating priorities

A priority level is said to be *active* when an exception of that priority is being handled: for interrupts, this is implied when the interrupt is acknowledged; for non-interrupt exceptions, such as SErrors or *SDEI explicit dispatches*, this has to be done via calling `ehf_activate_priority()`. See *Run-time flow*.

Conversely, when the dispatcher has reached a logical resolution for the cause of the exception, the corresponding priority level ought to be deactivated. As above, for interrupts, this is implied when the interrupt is EOId in the GIC; for other exceptions, this has to be done via calling `ehf_deactivate_priority()`.

Thanks to *different provisions* for exception delegation, there are potentially more than one work flow for deactivation:

- The dispatcher has addressed the cause of the exception, and decided to take no further action. In this case, the dispatcher's handler deactivates the priority level before returning to the *EHF*. Runtime firmware, upon exit through an `ERET`, resumes execution before the interrupt occurred.
- The dispatcher has to delegate the execution to lower ELs, and the cause of the exception can be considered resolved only when the lower EL returns signals complete (via an `SMC`) at a future point in time. The following sequence ensues:
  1. The dispatcher calls `setjmp()` to setup a jump point, and arranges to enter a lower EL upon the next `ERET`.
  2. Through the ensuing `ERET` from runtime firmware, execution is delegated to a lower EL.
  3. The lower EL completes its execution, and signals completion via an `SMC`.
  4. The `SMC` is handled by the same dispatcher that handled the exception previously. Noticing the conclusion of exception handling, the dispatcher does `longjmp()` to resume beyond the previous jump point.

As mentioned above, the *EHF* provides the following APIs for activating and deactivating interrupt:

- `ehf_activate_priority()` activates the supplied priority level, but only if the current active priority is higher than the given one; otherwise panics. Also, to prevent interruption by physical interrupts of lower priority, the *EHF* programs the *Priority Mask Register* corresponding to the PE to the priority being activated. Dispatchers typically only need to call this when handling exceptions other than interrupts, and it needs to delegate execution to a lower EL at a desired priority level.
- `ehf_deactivate_priority()` deactivates a given priority, but only if the current active priority is equal to the given one; otherwise panics. *EHF* also restores the *Priority Mask Register* corresponding to the PE to the priority before the call to `ehf_activate_priority()`. Dispatchers typically only need to call this after handling exceptions other than interrupts.

The calling of APIs are subject to allowed *transitions*. See also the *Run-time flow*.

4.4.7 Transition of priority levels

The *EHF* APIs `ehf_activate_priority()` and `ehf_deactivate_priority()` can be called to transition the current priority level on a PE. A given sequence of calls to these APIs are subject to the following conditions:

- For activation, the *EHF* only allows for the priority to increase (i.e. numeric value decreases);
- For deactivation, the *EHF* only allows for the priority to decrease (i.e. numeric value increases). Additionally, the priority being deactivated is required to be the current priority.

If these are violated, a panic will result.

4.4.8 Effect on SMC calls

In general, Secure execution is regarded as more important than Non-secure execution. As discussed elsewhere in this document, EL3 execution, and any delegated execution thereafter, has the effect of raising GIC's priority mask—either implicitly by acknowledging Secure interrupts, or when dispatchers call `ehf_activate_priority()`. As a result, Non-secure interrupts cannot preempt any Secure execution.

SMCs from Non-secure world are synchronous exceptions, and are mechanisms for Non-secure world to request Secure services. They're broadly classified as *Fast* or *Yielding* (see SMCCC).

- *Fast* SMCs are atomic from the caller's point of view. I.e., they return to the caller only when the Secure world has finished serving the request. Any Non-secure interrupts that become pending meanwhile cannot preempt Secure execution.

- *Yielding* SMCs carry the semantics of a preemptible, lower-priority request. A pending Non-secure interrupt can preempt Secure execution handling a Yielding SMC. I.e., the caller might observe a Yielding SMC returning when either:

  1. Secure world completes the request, and the caller would find `SMC_OK` as the return code.
  2. A Non-secure interrupt preempts Secure execution. Non-secure interrupt is handled, and Non-secure execution resumes after `SMC` instruction.

  The dispatcher handling a Yielding SMC must provide a different return code to the Non-secure caller to distinguish the latter case. This return code, however, is not standardised (unlike `SMC_UNKNOWN` or `SMC_OK`, for example), so will vary across dispatchers that handle the request.

For the latter case above, dispatchers before *EHF* expect Non-secure interrupts to be taken to S-EL1[2], so would get a chance to populate the designated preempted error code before yielding to Non-secure world.

The introduction of *EHF* changes the behaviour as described in *Interrupt handling*.

When *EHF* is enabled, in order to allow Non-secure interrupts to preempt Yielding SMC handling, the dispatcher must call `ehf_allow_ns_preemption()` API. The API takes one argument, the error code to be returned to the Non-secure world upon getting preempted.

4.4.9 Build-time flow

Please refer to the *figure* above.

The build-time flow involves the following steps:

1. Platform assigns priorities by installing priority level descriptors for individual dispatchers, as described in *Partitioning priority levels*.
2. Platform provides interrupt properties to GIC driver, as described in *Programming priority*.
3. Dispatcher calling `ehf_register_priority_handler()` to register an interrupt handler.

Also refer to the *Interrupt handling example*.

4.4.10 Run-time flow

The following is an example flow for interrupts:

1. The GIC driver, during initialization, iterates through the platform-supplied interrupt properties (see *Programming priority*), and configures the interrupts. This programs the appropriate priority and group (Group 0) on interrupts belonging to different dispatchers.
2. The *EHF*, during its initialisation, registers a top-level interrupt handler with the *Interrupt Management Framework* for EL3 interrupts. This also results in setting the routing bits in `SCR_EL3`.
3. When an interrupt belonging to a dispatcher fires, GIC raises an EL3/Group 0 interrupt, and is taken to EL3.
4. The top-level EL3 interrupt handler executes. The handler acknowledges the interrupt, reads its *Running Priority*, and from that, determines the dispatcher handler.
5. The *EHF* programs the *Priority Mask Register* of the PE to the priority of the interrupt received.
6. The *EHF* marks that priority level *active*, and jumps to the dispatcher handler.
7. Once the dispatcher handler finishes its job, it has to immediately *deactivate* the priority level before returning to the *EHF*. See *deactivation workflows*.

[2] In case of GICv2, Non-secure interrupts while in S-EL1 were signalled as IRQs, and in case of GICv3, FIQs.

The following is an example flow for exceptions that targets EL3 other than interrupt:

1. The platform provides handlers for the specific kind of exception.
2. The exception arrives, and the corresponding handler is executed.
3. The handler calls `ehf_activate_priority()` to activate the required priority level. This also has the effect of raising GIC priority mask, thus preventing interrupts of lower priority from preempting the handling. The handler may choose to do the handling entirely in EL3 or delegate to a lower EL.
4. Once exception handling concludes, the handler calls `ehf_deactivate_priority()` to deactivate the priority level activated earlier. This also has the effect of lowering GIC priority mask to what it was before.

4.4.11 Interrupt Prioritisation Considerations

The GIC priority scheme, by design, prioritises Secure interrupts over Normal world ones. The platform further assigns relative priorities amongst Secure dispatchers through *EHF*.

As mentioned in *Partitioning priority levels*, interrupts targeting distinct dispatchers fall in distinct priority levels. Because they're routed via the GIC, interrupt delivery to the PE is subject to GIC prioritisation rules. In particular, when an interrupt is being handled by the PE (i.e., the interrupt is in *Active* state), only interrupts of higher priority are signalled to the PE, even if interrupts of same or lower priority are pending. This has the side effect of one dispatcher being starved of interrupts by virtue of another dispatcher handling its (higher priority) interrupts.

The *EHF* doesn't enforce a particular prioritisation policy, but the platform should carefully consider the assignment of priorities to dispatchers integrated into runtime firmware. The platform should sensibly delineate priority to various dispatchers according to their nature. In particular, dispatchers of critical nature (RAS, for example) should be assigned higher priority than others (*SDEI*, for example); and within *SDEI*, Critical priority *SDEI* should be assigned higher priority than Normal ones.

4.4.12 Limitations

The *EHF* has the following limitations:

- Although there could be up to 128 Secure dispatchers supported by the GIC priority scheme, the size of descriptor array exposed with `EHF_REGISTER_PRIORITIES()` macro is currently limited to 32. This serves most expected use cases. This may be expanded in the future, should use cases demand so.
- The platform must ensure that the priority assigned to the dispatcher in the exception descriptor and the programmed priority of interrupts handled by the dispatcher match. The *EHF* cannot verify that this has been followed.

4.5 Firmware Configuration Framework

This document provides an overview of the *FCONF* framework.

4.5.1 Introduction

The Firmware CONfiguration Framework (*FCONF*) is an abstraction layer for platform specific data, allowing a "property" to be queried and a value retrieved without the requesting entity knowing what backing store is being used to hold the data.

It is used to bridge new and old ways of providing platform-specific data. Today, information like the Chain of Trust is held within several, nested platform-defined tables. In the future, it may be provided as part of a device blob, along with the rest of the information about images to load. Introducing this abstraction layer will make migration easier and will preserve functionality for platforms that cannot / don't want to use device tree.

4.5.2 Accessing properties

Properties defined in the *FCONF* are grouped around namespaces and sub-namespaces: a.b.property. Examples namespace can be:

- (*TBBR*) Chain of Trust data: tbbr.cot.trusted_boot_fw_cert
- (*TBBR*) dynamic configuration info: tbbr.dyn_config.disable_auth
- Arm io policies: arm.io_policies.bl2_image
- GICv3 properties: hw_config.gicv3_config.gicr_base

Properties can be accessed with the `FCONF_GET_PROPERTY(a,b,property)` macro.

4.5.3 Defining properties

Properties composing the *FCONF* have to be stored in C structures. If properties originate from a different backend source such as a device tree, then the platform has to provide a `populate()` function which essentially captures the property and stores them into a corresponding *FCONF* based C structure.

Such a `populate()` function is usually platform specific and is associated with a specific backend source. For example, a populator function which captures the hardware topology of the platform from the HW_CONFIG device tree. Hence each `populate()` function must be registered with a specific `config_type` identifier. It broadly represents a logical grouping of configuration properties which is usually a device tree file.

Example:

- FW_CONFIG: properties related to base address, maximum size and image id of other DTBs etc.
- TB_FW: properties related to trusted firmware such as IO policies, mbedtls heap info etc.
- HW_CONFIG: properties related to hardware configuration of the SoC such as topology, GIC controller, PSCI hooks, CPU ID etc.

Hence the `populate()` callback must be registered to the (*FCONF*) framework with the `FCONF_REGISTER_POPULATOR()` macro. This ensures that the function would be called inside the generic `fconf_populate()` function during initialization.

```
int fconf_populate_topology(uintptr_t config)
{
    /* read hw config dtb and fill soc_topology struct */
}

FCONF_REGISTER_POPULATOR(HW_CONFIG, topology, fconf_populate_topology);
```

Then, a wrapper has to be provided to match the `FCONF_GET_PROPERTY()` macro:

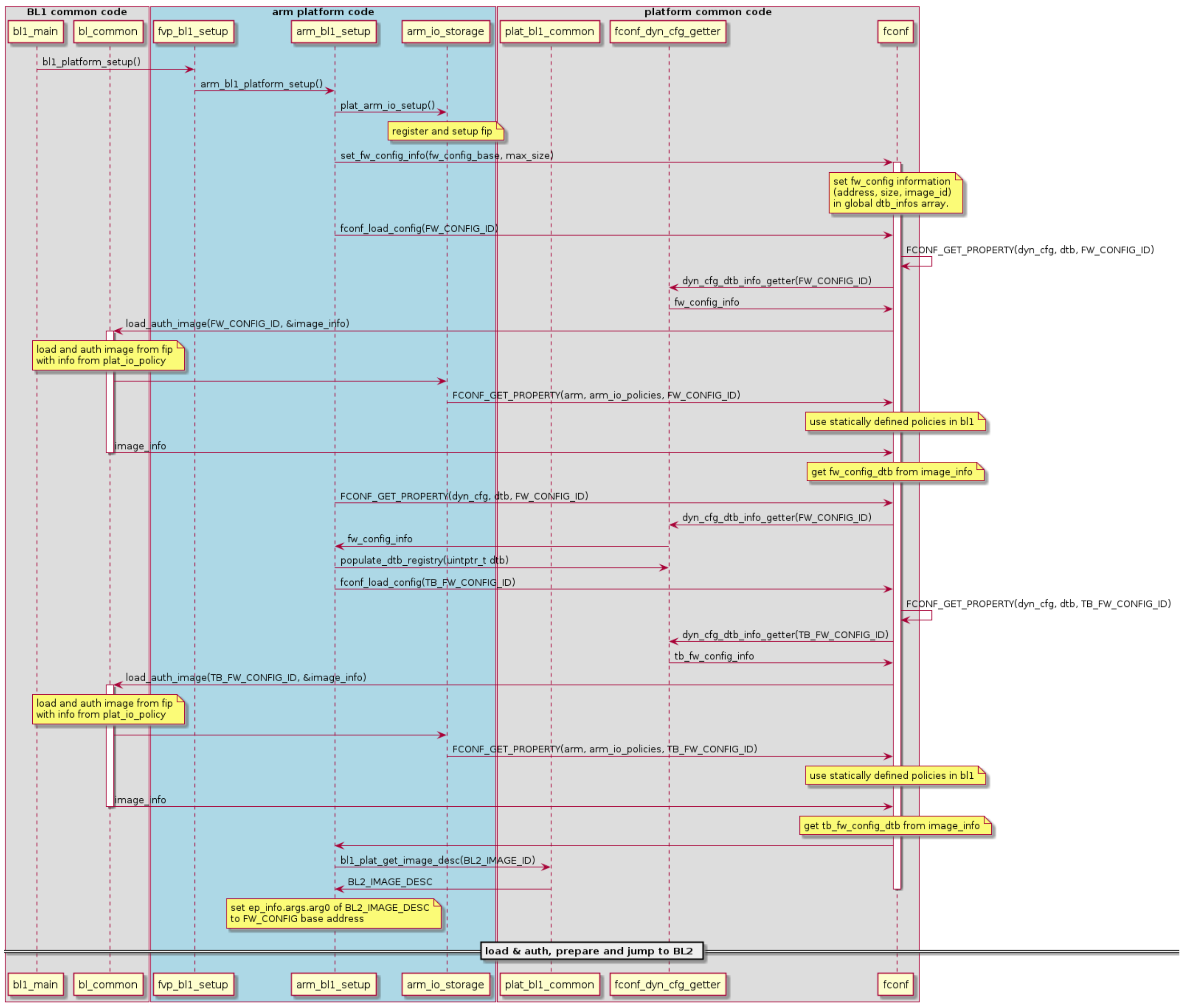
```
/* generic getter */
#define FCONF_GET_PROPERTY(a,b,property)    a##__##b##_getter(property)

/* my specific getter */
#define hw_config__topology_getter(prop) soc_topology.prop
```

This second level wrapper can be used to remap the `FCONF_GET_PROPERTY()` to anything appropriate: structure, array, function, etc..

To ensure a good interpretation of the properties, this documentation must explain how the properties are described for a specific backend. Refer to the *Properties binding information* section for more information and example.

4.5.4 Loading the property device tree

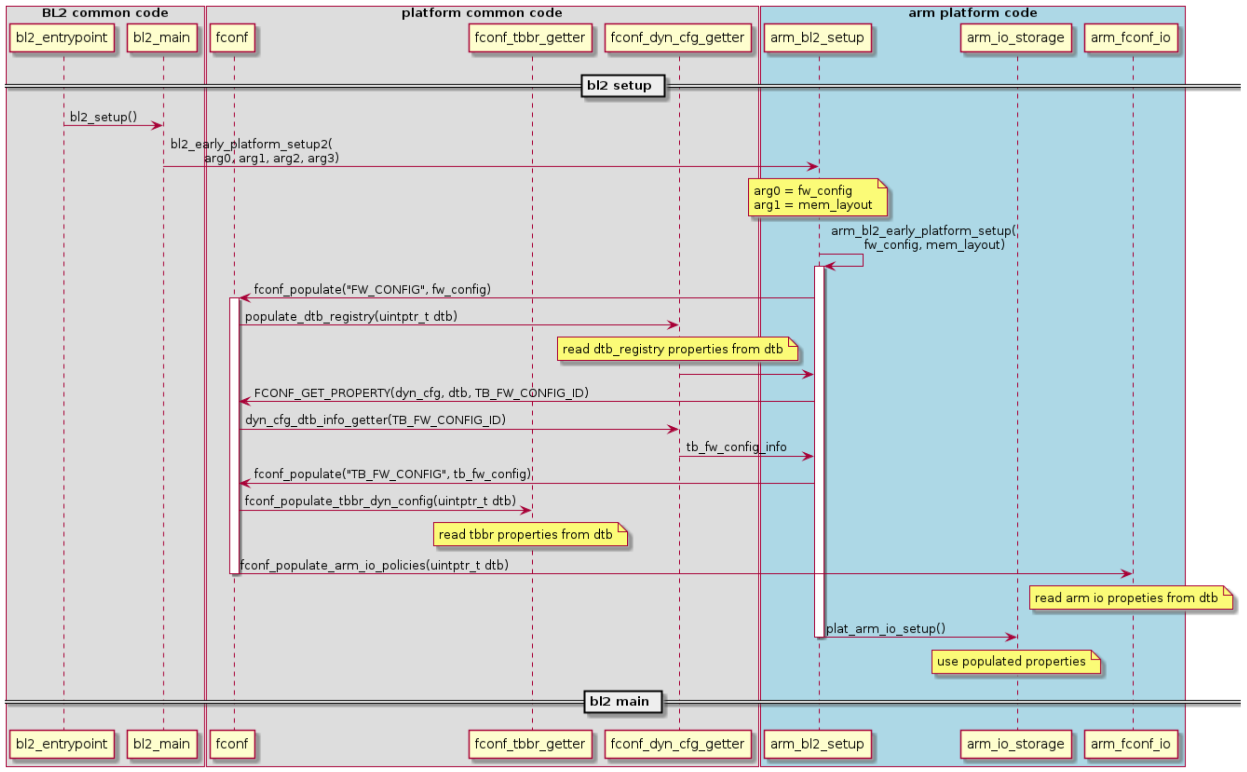
The `fconf_load_config(image_id)` must be called to load fw_config and tb_fw_config devices tree containing the properties' values. This must be done after the io layer is initialized, as the *DTB* is stored on an external device (FIP).



BL1 common code
arm platform code
platform common code
bl1_main
bl_common
fvp_bl1_setup
arm_bl1_setup
arm_io_storage
plat_bl1_common
fconf_dyn_cfg_getter
fconf
bl1_platform_setup()
arm_bl1_platform_setup()
plat_arm_io_setup()
register and setup fip
set_fw_config_info(fw_config_base, max_size)
set fw_config information (address, size, image_id) in global dtb_infos array.
fconf_load_config(FW_CONFIG_ID)
FCONF_GET_PROPERTY(dyn_cfg, dtb, FW_CONFIG_ID)
dyn_cfg_dtb_info_getter(FW_CONFIG_ID)
fw_config_info
load_auth_image(FW_CONFIG_ID, &image_info)
load and auth image from fip with info from plat_io_policy
FCONF_GET_PROPERTY(arm, arm_io_policies, FW_CONFIG_ID)
use statically defined policies in bl1
image_info
get fw_config_dtb from image_info
FCONF_GET_PROPERTY(dyn_cfg, dtb, FW_CONFIG_ID)
dyn_cfg_dtb_info_getter(FW_CONFIG_ID)
fw_config_info
populate_dtb_registry(uintptr_t dtb)
fconf_load_config(TB_FW_CONFIG_ID)
FCONF_GET_PROPERTY(dyn_cfg, dtb, TB_FW_CONFIG_ID)
dyn_cfg_dtb_info_getter(TB_FW_CONFIG_ID)
tb_fw_config_info
load_auth_image(TB_FW_CONFIG_ID, &image_info)
load and auth image from fip with info from plat_io_policy
FCONF_GET_PROPERTY(arm, arm_io_policies, TB_FW_CONFIG_ID)
use statically defined policies in bl1
image_info
get tb_fw_config_dtb from image_info
bl1_plat_get_image_desc(BL2_IMAGE_ID)
BL2_IMAGE_DESC
set ep_info.args.arg0 of BL2_IMAGE_DESC to FW_CONFIG base address
load & auth, prepare and jump to BL2

4.5.5 Populating the properties

Once a valid device tree is available, the `fconf_populate(config)` function can be used to fill the C data structure with the data from the config *DTB*. This function will call all the `populate()` callbacks which have been registered with `FCONF_REGISTER_POPULATOR()` as described above.


BL2 common code
platform common code
arm platform code
bl2_entrypoint
bl2_main
fconf
fconf_tbbr_getter
fconf_dyn_cfg_getter
arm_bl2_setup
arm_io_storage
arm_fconf_io
bl2 setup
bl2_setup()
bl2_early_platform_setup2(arg0, arg1, arg2, arg3)
arg0 = fw_config
arg1 = mem_layout
arm_bl2_early_platform_setup(fw_config, mem_layout)
fconf_populate("FW_CONFIG", fw_config)
populate_dtb_registry(uintptr_t dtb)
read dtb_registry properties from dtb
FCONF_GET_PROPERTY(dyn_cfg, dtb, TB_FW_CONFIG_ID)
dyn_cfg_dtb_info_getter(TB_FW_CONFIG_ID)
tb_fw_config_info
fconf_populate("TB_FW_CONFIG", tb_fw_config)
fconf_populate_tbbr_dyn_config(uintptr_t dtb)
read tbbr properties from dtb
fconf_populate_arm_io_policies(uintptr_t dtb)
read arm io propeties from dtb
plat_arm_io_setup()
use populated properties
bl2 main

4.5.6 Namespace guidance

As mentioned above, properties are logically grouped around namespaces and sub-namespaces. The following concepts should be considered when adding new properties/namespaces. The framework differentiates two types of properties:

- Properties used inside common code.
- Properties used inside platform specific code.

The first category applies to properties being part of the firmware and shared across multiple platforms. They should be globally accessible and defined inside the `lib/fconf` directory. The namespace must be chosen to reflect the feature/data abstracted.

Example:

- *TBBR* related properties: tbbr.cot.bl2_id
- Dynamic configuration information: dyn_cfg.dtb_info.hw_config_id

The second category should represent the majority of the properties defined within the framework: Platform specific properties. They must be accessed only within the platform API and are defined only inside the platform scope. The namespace must contain the platform name under which the properties defined belong.

Example:

- Arm io framework: arm.io_policies.bl31_id

4.5.7 Properties binding information

DTB binding for FCONF properties

This document describes the device tree format of *FCONF* properties. These properties are not related to a specific platform and can be queried from common code.

Dynamic configuration

The *FCONF* framework expects a *dtb-registry* node with the following field:

- **compatible [mandatory]**
  - value type: <string>
  - Must be the string "fconf,dyn_cfg-dtb_registry".

Then a list of subnodes representing a configuration *DTB*, which can be used by *FCONF*. Each subnode should be named according to the information it contains, and must be formed with the following fields:

- **load-address [mandatory]**
  - value type: <u64>
  - Physical loading base address of the configuration.
- **max-size [mandatory]**
  - value type: <u32>
  - Maximum size of the configuration.
- **id [mandatory]**
  - value type: <u32>
  - Image ID of the configuration.

4.6 Firmware Update (FWU)

4.6.1 Introduction

This document describes the design of the Firmware Update (FWU) feature, which enables authenticated firmware to update firmware images from external interfaces such as USB, UART, SD-eMMC, NAND, NOR or Ethernet to SoC Non-Volatile memories such as NAND Flash, LPDDR2-NVM or any memory determined by the platform. This feature functions even when the current firmware in the system is corrupt or missing; it therefore may be used as a recovery mode. It may also be complemented by other, higher level firmware update software.

FWU implements a specific part of the Trusted Board Boot Requirements (TBBR) specification, Arm DEN0006C-1. It should be used in conjunction with the *Trusted Board Boot* design document, which describes the image authentication parts of the Trusted Firmware-A (TF-A) TBBR implementation.

Scope

This document describes the secure world FWU design. It is beyond its scope to describe how normal world FWU images should operate. To implement normal world FWU images, please refer to the "Non-Trusted Firmware Updater" requirements in the TBBR.

4.6.2 FWU Overview

The FWU boot flow is primarily mediated by BL1. Since BL1 executes in ROM, and it is usually desirable to minimize the amount of ROM code, the design allows some parts of FWU to be implemented in other secure and normal world images. Platform code may choose which parts are implemented in which images but the general expectation is:

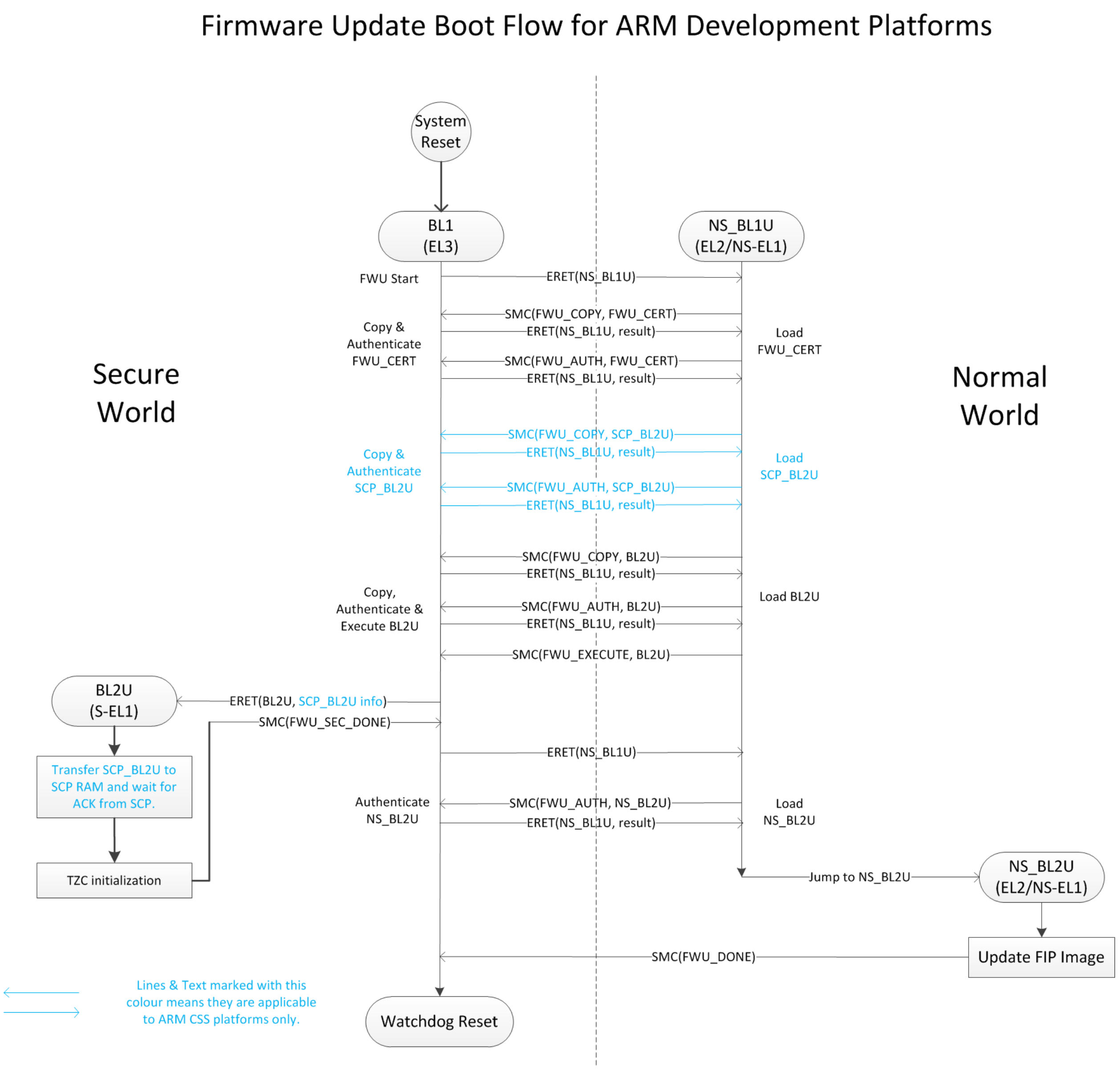
- BL1 handles:
  - Detection and initiation of the FWU boot flow.
  - Copying images from non-secure to secure memory
  - FWU image authentication
  - Context switching between the normal and secure world during the FWU process.
- Other secure world FWU images handle platform initialization required by the FWU process.
- Normal world FWU images handle loading of firmware images from external interfaces to non-secure memory.

The primary requirements of the FWU feature are:

1. Export a BL1 SMC interface to interoperate with other FWU images executing at other Exception Levels.
2. Export a platform interface to provide FWU common code with the information it needs, and to enable platform specific FWU functionality. See the *Porting Guide* for details of this interface.

TF-A uses abbreviated image terminology for FWU images like for other TF-A images. See the *Image Terminology* document for an explanation of these terms.

The following diagram shows the FWU boot flow for Arm development platforms. Arm CSS platforms like Juno have a System Control Processor (SCP), and these use all defined FWU images. Other platforms may use a subset of these.



4.6.3 Image Identification

Each FWU image and certificate is identified by a unique ID, defined by the platform, which BL1 uses to fetch an image descriptor (`image_desc_t`) via a call to `bl1_plat_get_image_desc()`. The same ID is also used to prepare the Chain of Trust (Refer to the *Authentication Framework & Chain of Trust* document for more information).

The image descriptor includes the following information:

- Executable or non-executable image. This indicates whether the normal world is permitted to request execution of a secure world FWU image (after authentication). Secure world certificates and non-AP images are examples of non-executable images.
- Secure or non-secure image. This indicates whether the image is authenticated/executed in secure or non-secure memory.
- Image base address and size.

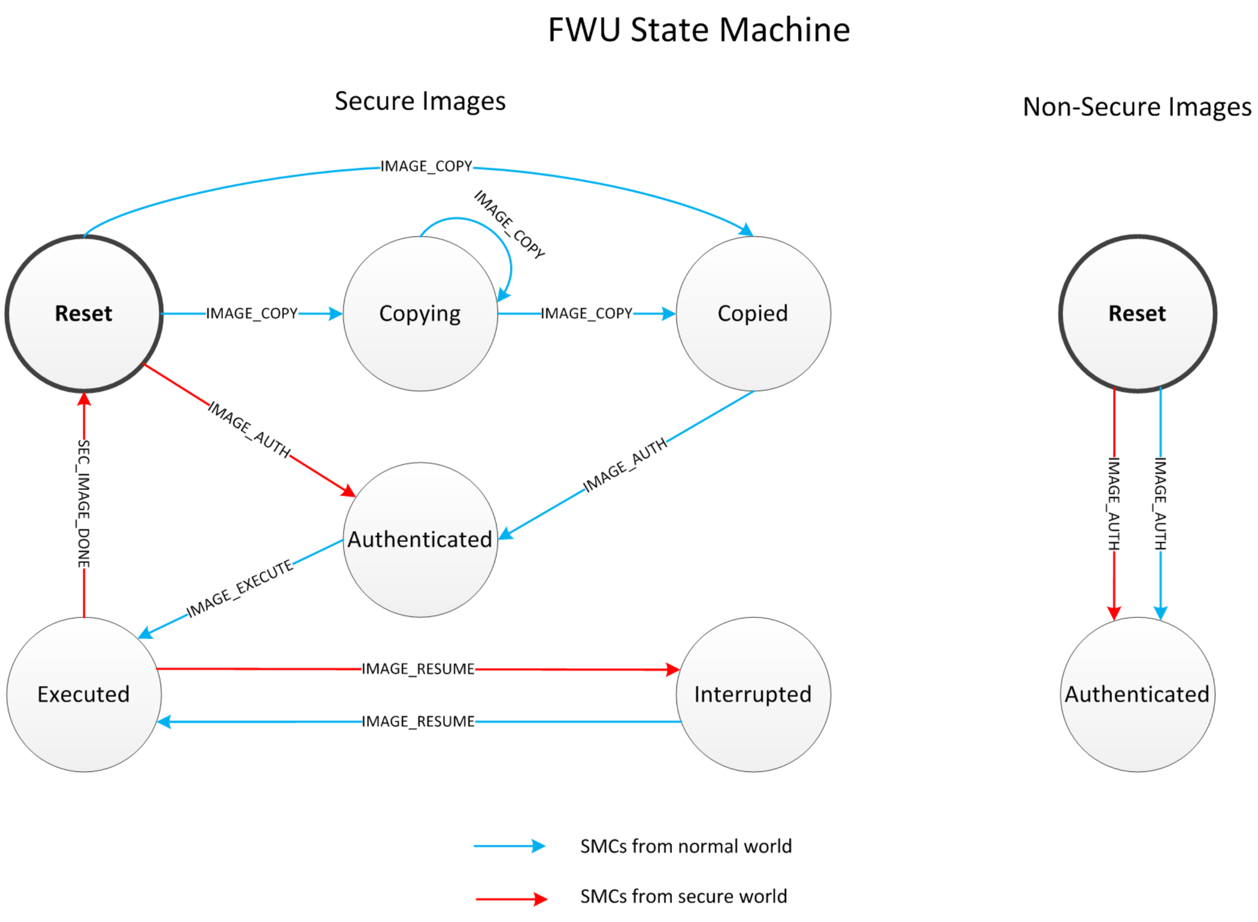
- Image entry point configuration (an `entry_point_info_t`).
- FWU image state.

BL1 uses the FWU image descriptors to:

- Validate the arguments of FWU SMCs
- Manage the state of the FWU process
- Initialize the execution state of the next FWU image.

4.6.4 FWU State Machine

BL1 maintains state for each FWU image during FWU execution. FWU images at lower Exception Levels raise SMCs to invoke FWU functionality in BL1, which causes BL1 to update its FWU image state. The BL1 image states and valid state transitions are shown in the diagram below. Note that secure images have a more complex state machine than non-secure images.



The following is a brief description of the supported states:

- RESET: This is the initial state of every image at the start of FWU. Authentication failure also leads to this state. A secure image may yield to this state if it has completed execution. It can also be reached by using `FWU_SMC_IMAGE_RESET`.
- COPYING: This is the state of a secure image while BL1 is copying it in blocks from non-secure to secure memory.

- COPIED: This is the state of a secure image when BL1 has completed copying it to secure memory.
- AUTHENTICATED: This is the state of an image when BL1 has successfully authenticated it.
- EXECUTED: This is the state of a secure, executable image when BL1 has passed execution control to it.
- INTERRUPTED: This is the state of a secure, executable image after it has requested BL1 to resume normal world execution.

4.6.5 BL1 SMC Interface

BL1_SMC_CALL_COUNT

```
Arguments:
    uint32_t function ID : 0x0

Return:
    uint32_t
```

This SMC returns the number of SMCs supported by BL1.

BL1_SMC_UID

```
Arguments:
    uint32_t function ID : 0x1

Return:
    UUID : 32 bits in each of w0-w3 (or r0-r3 for AArch32 callers)
```

This SMC returns the 128-bit Universally Unique Identifier for the BL1 SMC service.

BL1_SMC_VERSION

```
Argument:
    uint32_t function ID : 0x3

Return:
    uint32_t : Bits [31:16] Major Version
               Bits [15:0] Minor Version
```

This SMC returns the current version of the BL1 SMC service.

BL1_SMC_RUN_IMAGE

```
Arguments:
    uint32_t           function ID : 0x4
    entry_point_info_t *ep_info

Return:
    void

Pre-conditions:
```

(continues on next page)

(continued from previous page)

```
    if (normal world caller) synchronous exception
    if (ep_info not EL3) synchronous exception
```

This SMC passes execution control to an EL3 image described by the provided `entry_point_info_t` structure. In the normal TF-A boot flow, BL2 invokes this SMC for BL1 to pass execution control to BL31.

FWU_SMC_IMAGE_COPY

```
Arguments:
    uint32_t     function ID : 0x10
    unsigned int image_id
    uintptr_t    image_addr
    unsigned int block_size
    unsigned int image_size

Return:
    int : 0 (Success)
        : -ENOMEM
        : -EPERM

Pre-conditions:
    if (image_id is invalid) return -EPERM
    if (image_id is non-secure image) return -EPERM
    if (image_id state is not (RESET or COPYING)) return -EPERM
    if (secure world caller) return -EPERM
    if (image_addr + block_size overflows) return -ENOMEM
    if (image destination address + image_size overflows) return -ENOMEM
    if (source block is in secure memory) return -ENOMEM
    if (source block is not mapped into BL1) return -ENOMEM
    if (image_size > free secure memory) return -ENOMEM
    if (image overlaps another image) return -EPERM
```

This SMC copies the secure image indicated by `image_id` from non-secure memory to secure memory for later authentication. The image may be copied in a single block or multiple blocks. In either case, the total size of the image must be provided in `image_size` when invoking this SMC for the first time for each image; it is ignored in subsequent calls (if any) for the same image.

The `image_addr` and `block_size` specify the source memory block to copy from. The destination address is provided by the platform code.

If `block_size` is greater than the amount of remaining bytes to copy for this image then the former is truncated to the latter. The copy operation is then considered as complete and the FWU state machine transitions to the “COPIED” state. If there is still more to copy, the FWU state machine stays in or transitions to the COPYING state (depending on the previous state).

When using multiple blocks, the source blocks do not necessarily need to be in contiguous memory.

Once the SMC is handled, BL1 returns from exception to the normal world caller.

FWU_SMC_IMAGE_AUTH

```
Arguments:
    uint32_t     function ID : 0x11
    unsigned int image_id
    uintptr_t    image_addr
    unsigned int image_size

Return:
    int : 0 (Success)
        : -ENOMEM
        : -EPERM
        : -EAUTH

Pre-conditions:
    if (image_id is invalid) return -EPERM
    if (secure world caller)
        if (image_id state is not RESET) return -EPERM
        if (image_addr/image_size is not mapped into BL1) return -ENOMEM
    else // normal world caller
        if (image_id is secure image)
            if (image_id state is not COPIED) return -EPERM
        else // image_id is non-secure image
            if (image_id state is not RESET) return -EPERM
            if (image_addr/image_size is in secure memory) return -ENOMEM
            if (image_addr/image_size not mapped into BL1) return -ENOMEM
```

This SMC authenticates the image specified by `image_id`. If the image is in the RESET state, BL1 authenticates the image in place using the provided `image_addr` and `image_size`. If the image is a secure image in the COPIED state, BL1 authenticates the image from the secure memory that BL1 previously copied the image into.

BL1 returns from exception to the caller. If authentication succeeds then BL1 sets the image state to AUTHENTICATED. If authentication fails then BL1 returns the -EAUTH error and sets the image state back to RESET.

FWU_SMC_IMAGE_EXECUTE

```
Arguments:
    uint32_t     function ID : 0x12
    unsigned int image_id

Return:
    int : 0 (Success)
        : -EPERM

Pre-conditions:
    if (image_id is invalid) return -EPERM
    if (secure world caller) return -EPERM
    if (image_id is non-secure image) return -EPERM
    if (image_id is non-executable image) return -EPERM
    if (image_id state is not AUTHENTICATED) return -EPERM
```

This SMC initiates execution of a previously authenticated image specified by `image_id`, in the other security world to the caller. The current implementation only supports normal world callers initiating execution of a secure world image.

BL1 saves the normal world caller's context, sets the secure image state to EXECUTED, and returns from exception to the secure image.

FWU_SMC_IMAGE_RESUME

```
Arguments:
    uint32_t    function ID : 0x13
    register_t image_param

Return:
    register_t : image_param (Success)
               : -EPERM

Pre-conditions:
    if (normal world caller and no INTERRUPTED secure image) return -EPERM
```

This SMC resumes execution in the other security world while there is a secure image in the EXECUTED/INTERRUPTED state.

For normal world callers, BL1 sets the previously interrupted secure image state to EXECUTED. For secure world callers, BL1 sets the previously executing secure image state to INTERRUPTED. In either case, BL1 saves the calling world's context, restores the resuming world's context and returns from exception into the resuming world. If the call is successful then the caller provided `image_param` is returned to the resumed world, otherwise an error code is returned to the caller.

FWU_SMC_SEC_IMAGE_DONE

```
Arguments:
    uint32_t function ID : 0x14

Return:
    int : 0 (Success)
        : -EPERM

Pre-conditions:
    if (normal world caller) return -EPERM
```

This SMC indicates completion of a previously executing secure image.

BL1 sets the previously executing secure image state to the RESET state, restores the normal world context and returns from exception into the normal world.

FWU_SMC_UPDATE_DONE

```
Arguments:
    uint32_t    function ID : 0x15
    register_t client_cookie

Return:
    N/A
```

This SMC completes the firmware update process. BL1 calls the platform specific function `bl1_plat_fwu_done`, passing the optional argument `client_cookie` as a `void *`. The SMC does not return.

FWU_SMC_IMAGE_RESET

```
Arguments:
    uint32_t     function ID : 0x16
    unsigned int image_id

Return:
    int : 0 (Success)
        : -EPERM

Pre-conditions:
    if (secure world caller) return -EPERM
    if (image in EXECUTED) return -EPERM
```

This SMC sets the state of an image to RESET and zeroes the memory used by it.

This is only allowed if the image is not being executed.

4.7 Measured Boot Driver (MBD)

4.7.1 Properties binding information

DTB binding for Event Log properties

This document describes the device tree format of Event Log properties. These properties are not related to a specific platform and can be queried from common code.

Dynamic configuration for Event Log

Measured Boot driver expects a *tpm_event_log* node with the following field in 'nt_fw_config' and 'tsp_fw_config' DTS files:

- **compatible [mandatory]**
    - value type: <string>
    - Must be the string "arm,tpm_event_log".

Then a list of properties representing Event Log configuration, which can be used by Measured Boot driver. Each property is named according to the information it contains:

- **tpm_event_log_sm_addr [fvp_nt_fw_config.dts with OP-TEE]**
    - value type: <u64>
    - Event Log base address in secure memory.

Note. Currently OP-TEE does not support reading DTBs from Secure memory and this property should be removed when this feature is supported.

- **tpm_event_log_addr [mandatory]**
    - value type: <u64>

    - Event Log base address in non-secure memory.

- **tpm_event_log_size [mandatory]**
    - value type: <u32>
    - Event Log size.

4.8 Platform Interrupt Controller API

This document lists the optional platform interrupt controller API that abstracts the runtime configuration and control of interrupt controller from the generic code. The mandatory APIs are described in the *Porting Guide*.

4.8.1 Function: unsigned int plat_ic_get_running_priority(void); [optional]

```
Argument : void
Return   : unsigned int
```

This API should return the priority of the interrupt the PE is currently servicing. This must be be called only after an interrupt has already been acknowledged via `plat_ic_acknowledge_interrupt`.

In the case of Arm standard platforms using GIC, the *Running Priority Register* is read to determine the priority of the interrupt.

4.8.2 Function: int plat_ic_is_spi(unsigned int id); [optional]

```
Argument : unsigned int
Return   : int
```

The API should return whether the interrupt ID (first parameter) is categorized as a Shared Peripheral Interrupt. Shared Peripheral Interrupts are typically associated to system-wide peripherals, and these interrupts can target any PE in the system.

4.8.3 Function: int plat_ic_is_ppi(unsigned int id); [optional]

```
Argument : unsigned int
Return   : int
```

The API should return whether the interrupt ID (first parameter) is categorized as a Private Peripheral Interrupt. Private Peripheral Interrupts are typically associated with peripherals that are private to each PE. Interrupts from private peripherals target to that PE only.

4.8.4 Function: int plat_ic_is_sgi(unsigned int id); [optional]

```
Argument : unsigned int
Return   : int
```

The API should return whether the interrupt ID (first parameter) is categorized as a Software Generated Interrupt. Software Generated Interrupts are raised by explicit programming by software, and are typically used in inter-PE communication. Secure SGIs are reserved for use by Secure world software.

4.8.5 Function: unsigned int plat_ic_get_interrupt_active(unsigned int id); [optional]

```
Argument : unsigned int
Return   : int
```

This API should return the *active* status of the interrupt ID specified by the first parameter, `id`.

In case of Arm standard platforms using GIC, the implementation of the API reads the GIC *Set Active Register* to read and return the active status of the interrupt.

4.8.6 Function: void plat_ic_enable_interrupt(unsigned int id); [optional]

```
Argument : unsigned int
Return   : void
```

This API should enable the interrupt ID specified by the first parameter, `id`. PEs in the system are expected to receive only enabled interrupts.

In case of Arm standard platforms using GIC, the implementation of the API inserts barrier to make memory updates visible before enabling interrupt, and then writes to GIC *Set Enable Register* to enable the interrupt.

4.8.7 Function: void plat_ic_disable_interrupt(unsigned int id); [optional]

```
Argument : unsigned int
Return   : void
```

This API should disable the interrupt ID specified by the first parameter, `id`. PEs in the system are not expected to receive disabled interrupts.

In case of Arm standard platforms using GIC, the implementation of the API writes to GIC *Clear Enable Register* to disable the interrupt, and inserts barrier to make memory updates visible afterwards.

4.8.8 Function: void plat_ic_set_interrupt_priority(unsigned int id, unsigned int priority); [optional]

```
Argument : unsigned int
Argument : unsigned int
Return   : void
```

This API should set the priority of the interrupt specified by first parameter `id` to the value set by the second parameter `priority`.

In case of Arm standard platforms using GIC, the implementation of the API writes to GIC *Priority Register* set interrupt priority.

4.8.9 Function: int plat_ic_has_interrupt_type(unsigned int type); [optional]

```
Argument : unsigned int
Return   : int
```

This API should return whether the platform supports a given interrupt type. The parameter `type` shall be one of `INTR_TYPE_EL3`, `INTR_TYPE_S_EL1`, or `INTR_TYPE_NS`.

In case of Arm standard platforms using GICv3, the implementation of the API returns `1` for all interrupt types.

In case of Arm standard platforms using GICv2, the API always return `1` for `INTR_TYPE_NS`. Return value for other types depends on the value of build option `GICV2_G0_FOR_EL3`:

- For interrupt type `INTR_TYPE_EL3`:
  - When `GICV2_G0_FOR_EL3` is `0`, it returns `0`, indicating no support for EL3 interrupts.
  - When `GICV2_G0_FOR_EL3` is `1`, it returns `1`, indicating support for EL3 interrupts.
- For interrupt type `INTR_TYPE_S_EL1`:
  - When `GICV2_G0_FOR_EL3` is `0`, it returns `1`, indicating support for Secure EL1 interrupts.
  - When `GICV2_G0_FOR_EL3` is `1`, it returns `0`, indicating no support for Secure EL1 interrupts.

4.8.10 Function: void plat_ic_set_interrupt_type(unsigned int id, unsigned int type); [optional]

```
Argument : unsigned int
Argument : unsigned int
Return   : void
```

This API should set the interrupt specified by first parameter `id` to the type specified by second parameter `type`. The `type` parameter can be one of:

- `INTR_TYPE_NS`: interrupt is meant to be consumed by the Non-secure world.
- `INTR_TYPE_S_EL1`: interrupt is meant to be consumed by Secure EL1.
- `INTR_TYPE_EL3`: interrupt is meant to be consumed by EL3.

In case of Arm standard platforms using GIC, the implementation of the API writes to the GIC *Group Register* and *Group Modifier Register* (only GICv3) to assign the interrupt to the right group.

For GICv3:

- `INTR_TYPE_NS` maps to Group 1 interrupt.
- `INTR_TYPE_S_EL1` maps to Secure Group 1 interrupt.
- `INTR_TYPE_EL3` maps to Secure Group 0 interrupt.

For GICv2:

- `INTR_TYPE_NS` maps to Group 1 interrupt.
- When the build option `GICV2_G0_FOR_EL3` is set to `0` (the default), `INTR_TYPE_S_EL1` maps to Group 0. Otherwise, `INTR_TYPE_EL3` maps to Group 0 interrupt.

4.8.11 Function: void plat_ic_raise_el3_sgi(int sgi_num, u_register_t target); [optional]

```
Argument : int
Argument : u_register_t
Return   : void
```

This API should raise an EL3 SGI. The first parameter, `sgi_num`, specifies the ID of the SGI. The second parameter, `target`, must be the MPIDR of the target PE.

In case of Arm standard platforms using GIC, the implementation of the API inserts barrier to make memory updates visible before raising SGI, then writes to appropriate *SGI Register* in order to raise the EL3 SGI.

4.8.12 Function: void plat_ic_set_spi_routing(unsigned int id, unsigned int routing_mode, u_register_t mpidr); [optional]

```
Argument : unsigned int
Argument : unsigned int
Argument : u_register_t
Return   : void
```

This API should set the routing mode of Share Peripheral Interrupt (SPI) specified by first parameter `id` to that specified by the second parameter `routing_mode`.

The `routing_mode` parameter can be one of:

- `INTR_ROUTING_MODE_ANY` means the interrupt can be routed to any PE in the system. The `mpidr` parameter is ignored in this case.
- `INTR_ROUTING_MODE_PE` means the interrupt is routed to the PE whose MPIDR value is specified by the parameter `mpidr`.

In case of Arm standard platforms using GIC, the implementation of the API writes to the GIC *Target Register* (GICv2) or *Route Register* (GICv3) to set the routing.

4.8.13 Function: void plat_ic_set_interrupt_pending(unsigned int id); [optional]

```
Argument : unsigned int
Return   : void
```

This API should set the interrupt specified by first parameter `id` to *Pending*.

In case of Arm standard platforms using GIC, the implementation of the API inserts barrier to make memory updates visible before setting interrupt pending, and writes to the GIC *Set Pending Register* to set the interrupt pending status.

4.8.14 Function: void plat_ic_clear_interrupt_pending(unsigned int id); [optional]

```
Argument : unsigned int
Return   : void
```

This API should clear the *Pending* status of the interrupt specified by first parameter `id`.

In case of Arm standard platforms using GIC, the implementation of the API writes to the GIC *Clear Pending Register* to clear the interrupt pending status, and inserts barrier to make memory updates visible afterwards.

4.8.15 Function: unsigned int plat_ic_set_priority_mask(unsigned int id); [optional]

```
Argument : unsigned int
Return   : int
```

This API should set the priority mask (first parameter) in the interrupt controller such that only interrupts of higher priority than the supplied one may be signalled to the PE. The API should return the current priority value that it's overwriting.

In case of Arm standard platforms using GIC, the implementation of the API inserts to order memory updates before updating mask, then writes to the GIC *Priority Mask Register*, and make sure memory updates are visible before potential trigger due to mask update.

4.8.16 Function: unsigned int plat_ic_get_interrupt_id(unsigned int raw); [optional]

```
Argument : unsigned int
Return   : unsigned int
```

This API should extract and return the interrupt number from the raw value obtained by the acknowledging the interrupt (read using `plat_ic_acknowledge_interrupt()`). If the interrupt ID is invalid, this API should return `INTR_ID_UNAVAILABLE`.

In case of Arm standard platforms using GIC, the implementation of the API masks out the interrupt ID field from the acknowledged value from GIC.

4.9 Reliability, Availability, and Serviceability (RAS) Extensions

This document describes *TF-A* support for Arm Reliability, Availability, and Serviceability (RAS) extensions. RAS is a mandatory extension for Armv8.2 and later CPUs, and also an optional extension to the base Armv8.0 architecture.

In conjunction with the *EHF*, support for RAS extension enables firmware-first paradigm for handling platform errors: exceptions resulting from errors are routed to and handled in EL3. Said errors are Synchronous External Abort (SEA), Asynchronous External Abort (signalled as SErrors), Fault Handling and Error Recovery interrupts. The *EHF* document mentions various *error handling use-cases* .

For the description of Arm RAS extensions, Standard Error Records, and the precise definition of RAS terminology, please refer to the Arm Architecture Reference Manual. The rest of this document assumes familiarity with architecture and terminology.

4.9.1 Overview

As mentioned above, the RAS support in *TF-A* enables routing to and handling of exceptions resulting from platform errors in EL3. It allows the platform to define an External Abort handler, and to register RAS nodes and interrupts. RAS framework also provides *helpers* for accessing Standard Error Records as introduced by the RAS extensions.

The build option `RAS_EXTENSION` when set to `1` includes the RAS in run time firmware; `EL3_EXCEPTION_HANDLING` and `HANDLE_EA_EL3_FIRST` must also be set `1`. `RAS_TRAP_LOWER_EL_ERR_ACCESS` controls the access to the RAS error record registers from lower ELs.

See more on *Engaging the RAS framework*.

4.9.2 Platform APIs

The RAS framework allows the platform to define handlers for External Abort, Uncontainable Errors, Double Fault, and errors rising from EL3 execution. Please refer to *RAS Porting Guide*.

4.9.3 Registering RAS error records

RAS nodes are components in the system capable of signalling errors to PEs through one one of the notification mechanisms—SEAs, SErrors, or interrupts. RAS nodes contain one or more error records, which are registers through which the nodes advertise various properties of the signalled error. Arm recommends that error records are implemented in the Standard Error Record format. The RAS architecture allows for error records to be accessible via system or memory-mapped registers.

The platform should enumerate the error records providing for each of them:

- A handler to probe error records for errors;
- When the probing identifies an error, a handler to handle it;
- For memory-mapped error record, its base address and size in KB; for a system register-accessed record, the start index of the record and number of continuous records from that index;
- Any node-specific auxiliary data.

With this information supplied, when the run time firmware receives one of the notification mechanisms, the RAS framework can iterate through and probe error records for error, and invoke the appropriate handler to handle it.

The RAS framework provides the macros to populate error record information. The macros are versioned, and the latest version as of this writing is 1. These macros create a structure of type `struct err_record_info` from its arguments, which are later passed to probe and error handlers.

For memory-mapped error records:

```
ERR_RECORD_MEMMAP_V1(base_addr, size_num_k, probe, handler, aux)
```

And, for system register ones:

```
ERR_RECORD_SYSREG_V1(idx_start, num_idx, probe, handler, aux)
```

The probe handler must have the following prototype:

```
typedef int (*err_record_probe_t)(const struct err_record_info *info,
                int *probe_data);
```

The probe handler must return a non-zero value if an error was detected, or 0 otherwise. The `probe_data` output parameter can be used to pass any useful information resulting from probe to the error handler (see *below*). For example, it could return the index of the record.

The error handler must have the following prototype:

```
typedef int (*err_record_handler_t)(const struct err_record_info *info,
           int probe_data, const struct err_handler_data *const data);
```

The `data` constant parameter describes the various properties of the error, including the reason for the error, exception syndrome, and also `flags`, `cookie`, and `handle` parameters from the *top-level exception handler*.

The platform is expected populate an array using the macros above, and register the it with the RAS framework using the macro `REGISTER_ERR_RECORD_INFO()`, passing it the name of the array describing the records. Note that the macro must be used in the same file where the array is defined.

Standard Error Record helpers

The *TF-A* RAS framework provides probe handlers for Standard Error Records, for both memory-mapped and System Register accesses:

```
int ras_err_ser_probe_memmap(const struct err_record_info *info,
            int *probe_data);

int ras_err_ser_probe_sysreg(const struct err_record_info *info,
            int *probe_data);
```

When the platform enumerates error records, for those records in the Standard Error Record format, these helpers maybe used instead of rolling out their own. Both helpers above:

- Return non-zero value when an error is detected in a Standard Error Record;
- Set `probe_data` to the index of the error record upon detecting an error.

4.9.4 Registering RAS interrupts

RAS nodes can signal errors to the PE by raising Fault Handling and/or Error Recovery interrupts. For the firmware-first handling paradigm for interrupts to work, the platform must setup and register with *EHF*. See *Interaction with Exception Handling Framework*.

For each RAS interrupt, the platform has to provide structure of type `struct ras_interrupt`:

- Interrupt number;
- The associated error record information (pointer to the corresponding `struct err_record_info`);
- Optionally, a cookie.

The platform is expected to define an array of `struct ras_interrupt`, and register it with the RAS framework using the macro `REGISTER_RAS_INTERRUPTS()`, passing it the name of the array. Note that the macro must be used in the same file where the array is defined.

The array of `struct ras_interrupt` must be sorted in the increasing order of interrupt number. This allows for fast look of handlers in order to service RAS interrupts.

4.9.5 Double-fault handling

A Double Fault condition arises when an error is signalled to the PE while handling of a previously signalled error is still underway. When a Double Fault condition arises, the Arm RAS extensions only require for handler to perform orderly shutdown of the system, as recovery may be impossible.

The RAS extensions part of Armv8.4 introduced new architectural features to deal with Double Fault conditions, specifically, the introduction of `NMEA` and `EASE` bits to `SCR_EL3` register. These were introduced to assist EL3 software which runs part of its entry/exit routines with exceptions momentarily masked—meaning, in such systems, External Aborts/SErrors are not immediately handled when they occur, but only after the exceptions are unmasked again.

TF-A, for legacy reasons, executes entire EL3 with all exceptions unmasked. This means that all exceptions routed to EL3 are handled immediately. *TF-A* thus is able to detect a Double Fault conditions in software, without needing the intended advantages of Armv8.4 Double Fault architecture extensions.

Double faults are fatal, and terminate at the platform double fault handler, and doesn't return.

4.9.6 Engaging the RAS framework

Enabling RAS support is a platform choice constructed from three distinct, but related, build options:

- `RAS_EXTENSION=1` includes the RAS framework in the run time firmware;
- `EL3_EXCEPTION_HANDLING=1` enables handling of exceptions at EL3. See *Interaction with Exception Handling Framework*;
- `HANDLE_EA_EL3_FIRST=1` enables routing of External Aborts and SErrors to EL3.

The RAS support in *TF-A* introduces a default implementation of `plat_ea_handler`, the External Abort handler in EL3. When `RAS_EXTENSION` is set to `1`, it'll first call `ras_ea_handler()` function, which is the top-level RAS exception handler. `ras_ea_handler` is responsible for iterating to through platform-supplied error records, probe them, and when an error is identified, look up and invoke the corresponding error handler.

Note that, if the platform chooses to override the `plat_ea_handler` function and intend to use the RAS framework, it must explicitly call `ras_ea_handler()` from within.

Similarly, for RAS interrupts, the framework defines `ras_interrupt_handler()`. The RAS framework arranges for it to be invoked when a RAS interrupt taken at EL3. The function bisects the platform-supplied sorted array of interrupts to look up the error record information associated with the interrupt number. That error handler for that record is then invoked to handle the error.

4.9.7 Interaction with Exception Handling Framework

As mentioned in earlier sections, RAS framework interacts with the *EHF* to arbitrate handling of RAS exceptions with others that are routed to EL3. This means that the platform must partition a *priority level* for handling RAS exceptions. The platform must then define the macro `PLAT_RAS_PRI` to the priority level used for RAS exceptions. Platforms would typically want to allocate the highest secure priority for RAS handling.

Handling of both *interrupt* and *non-interrupt* exceptions follow the sequences outlined in the *EHF* documentation. I.e., for interrupts, the priority management is implicit; but for non-interrupt exceptions, they're explicit using *EHF APIs*.

4.10 Library at ROM

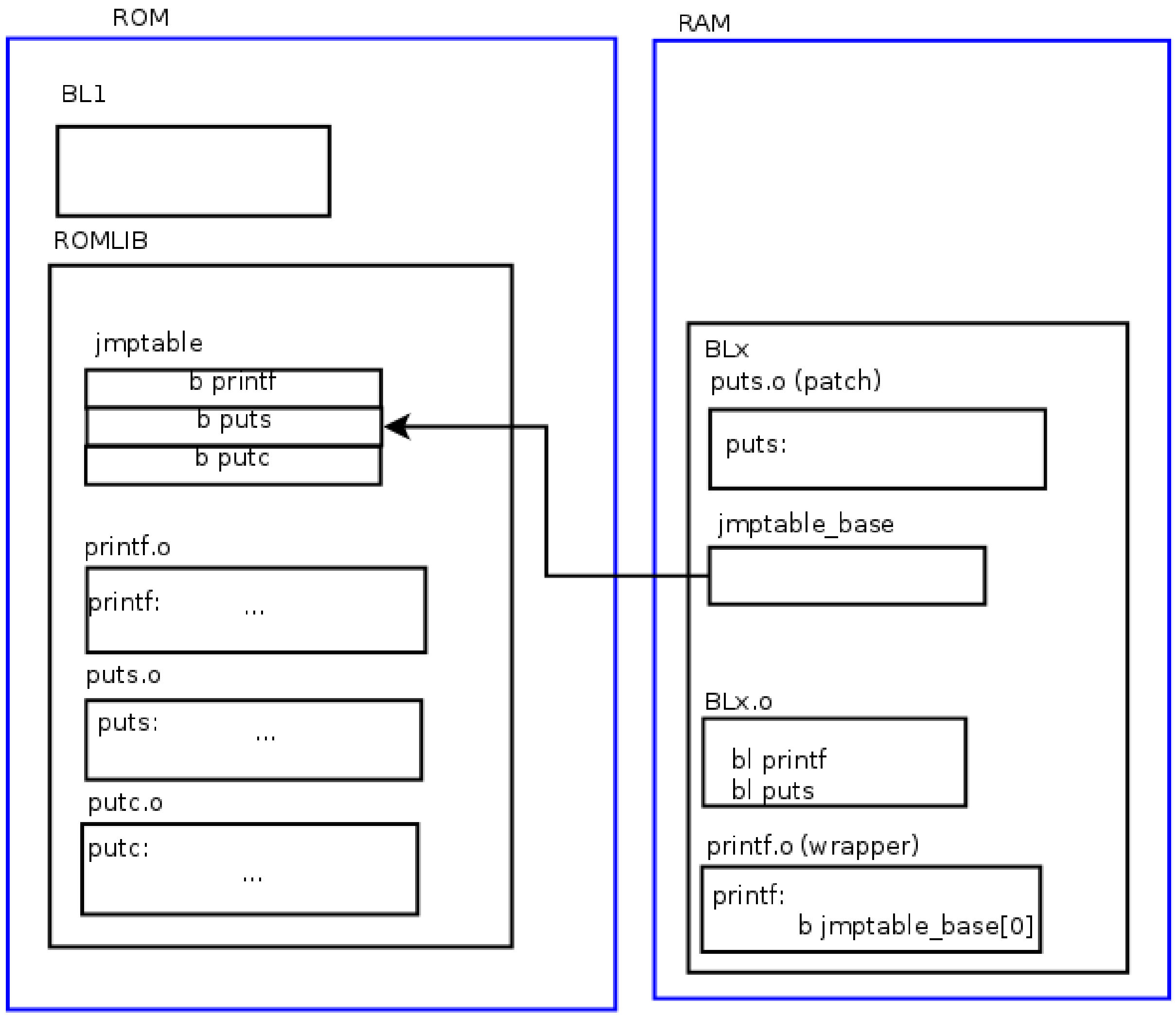
This document provides an overview of the "library at ROM" implementation in Trusted Firmware-A (TF-A).

4.10.1 Introduction

The "library at ROM" feature allows platforms to build a library of functions to be placed in ROM. This reduces SRAM usage by utilising the available space in ROM. The "library at ROM" contains a jump table with the list of functions that are placed in ROM. The capabilities of the "library at ROM" are:

1. Functions can be from one or several libraries.
2. Functions can be patched after they have been programmed into ROM.
3. Platform-specific libraries can be placed in ROM.
4. Functions can be accessed by one or more BL images.

4.10.2 Index file



Library at ROM is described by an index file with the list of functions to be placed in ROM. The index file is platform specific and its format is:

```
lib function    [patch]

lib      -- Name of the library the function belongs to
function -- Name of the function to be placed in library at ROM
[patch]  -- Option to patch the function
```

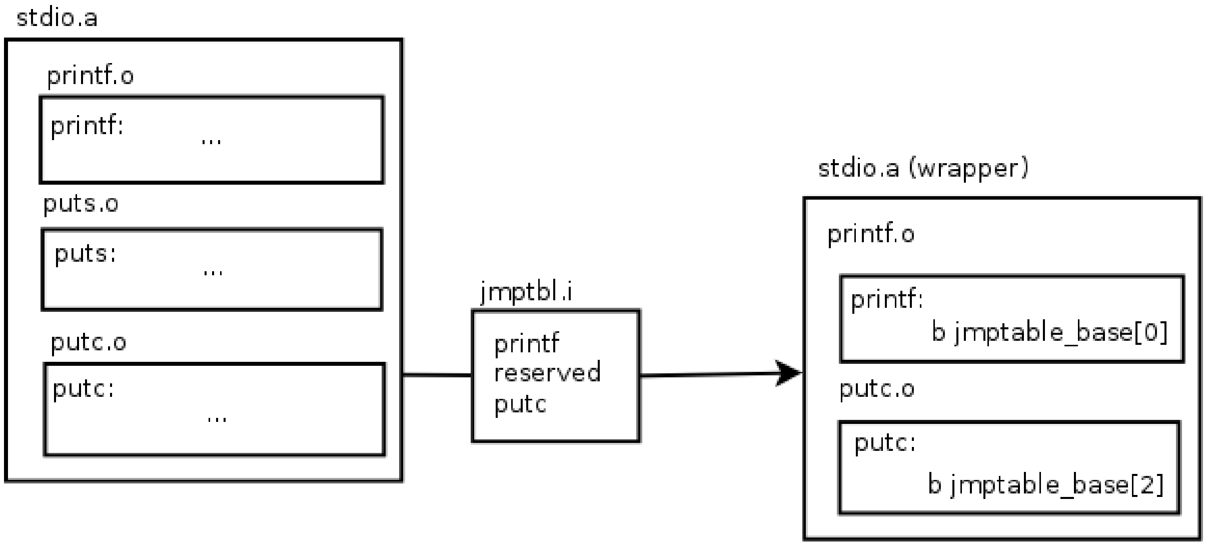
It is also possible to insert reserved spaces in the list by using the keyword "reserved" rather than the "lib" and "function" names as shown below:

```
reserved
```

The reserved spaces can be used to add more functions in the future without affecting the order and location of functions already existing in the jump table. Also, for additional flexibility and modularity, the index file can include other index files.

For an index file example, refer to `lib/romlib/jmptbl.i`.

4.10.3 Wrapper functions



When invoking a function of the "library at ROM", the calling sequence is as follows:

BL image –> wrapper function –> jump table entry –> library at ROM

The index file is used to create a jump table which is placed in ROM. Then, the wrappers refer to the jump table to call the "library at ROM" functions. The wrappers essentially contain a branch instruction to the jump table entry corresponding to the original function. Finally, the original function in the BL image(s) is replaced with the wrapper function.

The "library at ROM" contains a necessary init function that initialises the global variables defined by the functions inside "library at ROM".

4.10.4 Script

There is a `romlib_generate.py` Python script that generates the necessary files for the "library at ROM" to work. It implements multiple functions:

1. `romlib_generate.py gentbl [args]` - Generates the jump table by parsing the index file.
2. `romlib_generator.py genvar [args]` - Generates the jump table global variable (**not** the jump table itself) with the absolute address in ROM. This global variable is, basically, a pointer to the jump table.
3. `romlib_generator.py genwrappers [args]` - Generates a wrapper function for each entry in the index file except for the ones that contain the keyword `patch`. The generated wrapper file is called `<fn_name>.s`.
4. `romlib_generator.py pre [args]` - Preprocesses the index file which means it resolves all the include commands in the file recursively. It can also generate a dependency file of the included index files which can be directly used in makefiles.

Each `romlib_generate.py` function has its own manual which is accessible by runing `romlib_generator.py [function] --help`.

`romlib_generate.py` requires Python 3 environment.

4.10.5 Patching of functions in library at ROM

The `romlib_generator.py genwrappers` does not generate wrappers for the entries in the index file that contain the keyword `patch`. Thus, it allows calling the function from the actual library by breaking the link to the "library at ROM" version of this function.

The calling sequence for a patched function is as follows:

BL image –> function

4.10.6 Memory impact

Using library at ROM will modify the memory layout of the BL images:

- The ROM library needs a page aligned RAM section to hold the RW data. This section is defined by the ROMLIB_RW_BASE and ROMLIB_RW_END macros. On Arm platforms a section of 1 page (0x1000) is allocated at the top of SRAM. This will have for effect to shift down all the BL images by 1 page.
- Depending on the functions moved to the ROM library, the size of the BL images will be reduced. For example: moving MbedTLS function into the ROM library reduces BL1 and BL2, but not BL31.
- This change in BL images size can be taken into consideration to optimize the memory layout when defining the BLx_BASE macros.

4.10.7 Build library at ROM

The environment variable `CROSS_COMPILE` must be set appropriately. Refer to *Performing an Initial Build* for more information about setting this variable.

In the below example the usage of ROMLIB together with mbed TLS is demonstrated to showcase the benefits of library at ROM - it's not mandatory.

```
make PLAT=fvp                                                 \
MBEDTLS_DIR=</path/to/mbedtls/>                               \
TRUSTED_BOARD_BOOT=1 GENERATE_COT=1                           \
ARM_ROTPK_LOCATION=devel_rsa                                  \
ROT_KEY=plat/arm/board/common/rotpk/arm_rotprivk_rsa.pem      \
BL33=</path/to/bl33.bin>                                      \
USE_ROMLIB=1                                                  \
all fip
```

4.11 SDEI: Software Delegated Exception Interface

This document provides an overview of the SDEI dispatcher implementation in Trusted Firmware-A (TF-A).

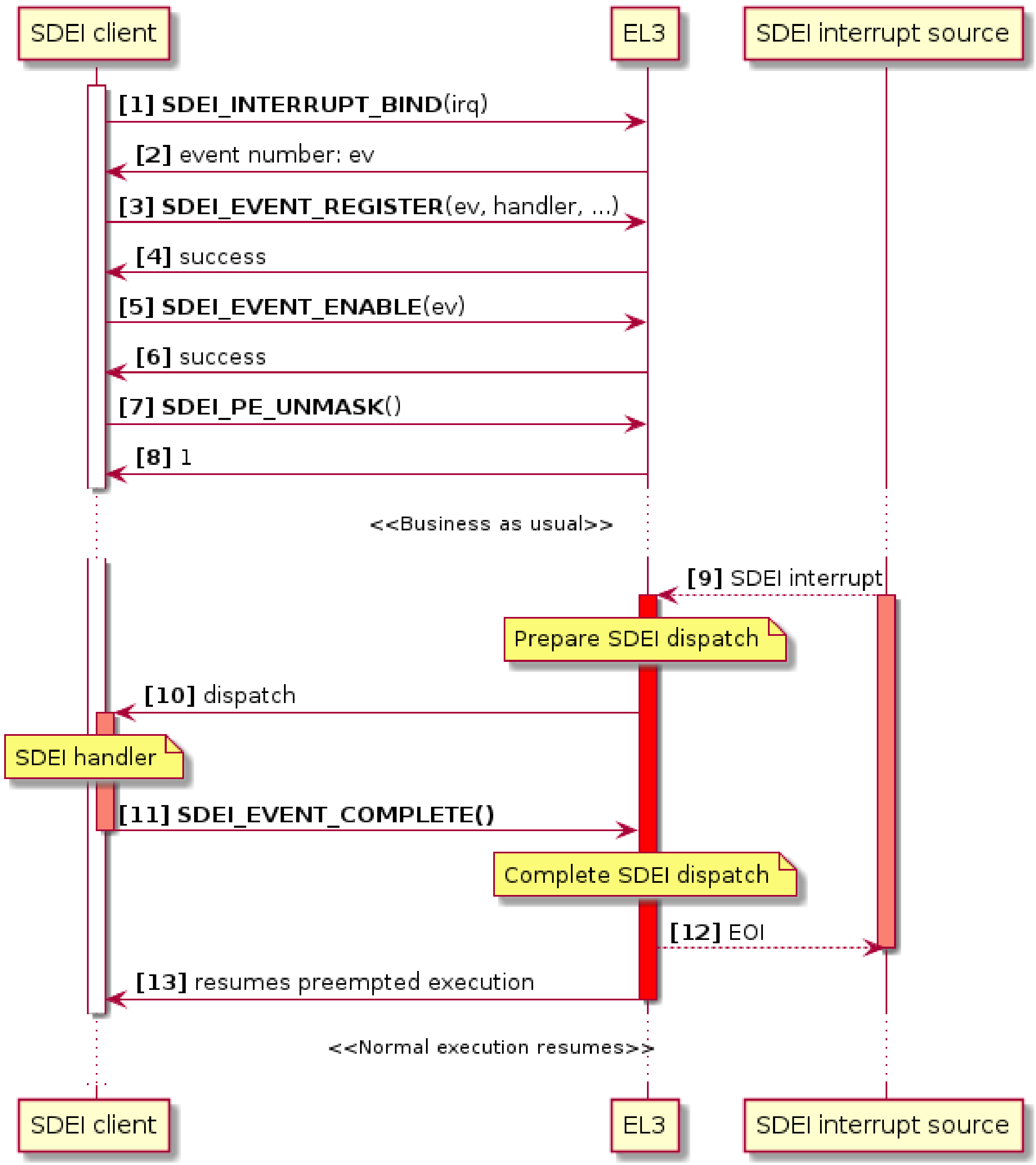
4.11.1 Introduction

Software Delegated Exception Interface (*SDEI*) is an Arm specification for Non-secure world to register handlers with firmware to receive notifications about system events. Firmware will first receive the system events by way of asynchronous exceptions and, in response, arranges for the registered handler to execute in the Non-secure EL.

Normal world software that interacts with the SDEI dispatcher (makes SDEI requests and receives notifications) is referred to as the *SDEI Client*. A client receives the event notification at the registered handler even when it was executing with exceptions masked. The list of SDEI events available to the client are specific to the platform[1]. See also *Determining client EL*.

The following figure depicts a general sequence involving SDEI client executing at EL2 and an event dispatch resulting from the triggering of a bound interrupt. A commentary is provided below:

[1] Except event 0, which is defined by the SDEI specification as a standard event.



As part of initialisation, the SDEI client binds a Non-secure interrupt [1], and the SDEI dispatcher returns a platform dynamic event number [2]. The client then registers a handler for that event [3], enables the event [5], and unmasks all events on the current PE [7]. This sequence is typical of an SDEI client, but it may involve additional SDEI calls.

At a later point in time, when the bound interrupt triggers [9], it's trapped to EL3. The interrupt is handed over to the SDEI dispatcher, which then arranges to execute the registered handler [10]. The client terminates its execution with `SDEI_EVENT_COMPLETE` [11], following which the dispatcher resumes the original EL2 execution [13]. Note that the SDEI interrupt remains active until the client handler completes, at which point EL3 does EOI [12].

Other than events bound to interrupts, as depicted in the sequence above, SDEI events can be explicitly dispatched in response to other exceptions, for example, upon receiving an *SError* or *Synchronous External Abort*. See *Explicit* *dispatch of events*.

The remainder of this document only discusses the design and implementation of SDEI dispatcher in TF-A, and assumes that the reader is familiar with the SDEI specification, the interfaces, and their requirements.

4.11.2 Defining events

A platform choosing to include the SDEI dispatcher must also define the events available on the platform, along with their attributes.

The platform is expected to provide two arrays of event descriptors: one for private events, and another for shared events. The SDEI dispatcher provides `SDEI_PRIVATE_EVENT()` and `SDEI_SHARED_EVENT()` macros to populate the event descriptors. Both macros take 3 arguments:

- The event number: this must be a positive 32-bit integer.
- For an event that has a backing interrupt, the interrupt number the event is bound to:
  - If it's not applicable to an event, this shall be left as `0`.
  - If the event is dynamic, this should be specified as `SDEI_DYN_IRQ`.
- A bit map of *Event flags*.

To define event 0, the macro `SDEI_DEFINE_EVENT_0()` should be used. This macro takes only one parameter: an SGI number to signal other PEs.

To define an event that's meant to be explicitly dispatched (i.e., not as a result of receiving an SDEI interrupt), the macro `SDEI_EXPLICIT_EVENT()` should be used. It accepts two parameters:

- The event number (as above);
- Event priority: `SDEI_MAPF_CRITICAL` or `SDEI_MAPF_NORMAL`, as described below.

Once the event descriptor arrays are defined, they should be exported to the SDEI dispatcher using the `REGISTER_SDEI_MAP()` macro, passing it the pointers to the private and shared event descriptor arrays, respectively. Note that the `REGISTER_SDEI_MAP()` macro must be used in the same file where the arrays are defined.

Regarding event descriptors:

- For Event 0:
  - There must be exactly one descriptor in the private array, and none in the shared array.
  - The event should be defined using `SDEI_DEFINE_EVENT_0()`.
  - Must be bound to a Secure SGI on the platform.
- Explicit events should only be used in the private array.
- Statically bound shared and private interrupts must be bound to shared and private interrupts on the platform, respectively. See the section on *Configuration within Exception Handling Framework*.
- Both arrays should be one-dimensional. The `REGISTER_SDEI_MAP()` macro takes care of replicating private events for each PE on the platform.
- Both arrays must be sorted in the increasing order of event number.

The SDEI specification doesn't have provisions for discovery of available events on the platform. The list of events made available to the client, along with their semantics, have to be communicated out of band; for example, through Device Trees or firmware configuration tables.

See also *Event definition example*.

Event flags

Event flags describe the properties of the event. They are bit maps that can be ORed to form parameters to macros that define events (see *Defining events*).

- SDEI_MAPF_DYNAMIC: Marks the event as dynamic. Dynamic events can be bound to (or released from) any Non-secure interrupt at runtime via the SDEI_INTERRUPT_BIND and SDEI_INTERRUPT_RELEASE calls.
- SDEI_MAPF_BOUND: Marks the event as statically bound to an interrupt. These events cannot be re-bound at runtime.
- SDEI_MAPF_NORMAL: Marks the event as having *Normal* priority. This is the default priority.
- SDEI_MAPF_CRITICAL: Marks the event as having *Critical* priority.

4.11.3 Event definition example

```
static sdei_ev_map_t plat_private_sdei[] = {
     /* Event 0 definition */
     SDEI_DEFINE_EVENT_0(8),

     /* PPI */
     SDEI_PRIVATE_EVENT(8, 23, SDEI_MAPF_BOUND),

     /* Dynamic private events */
     SDEI_PRIVATE_EVENT(100, SDEI_DYN_IRQ, SDEI_MAPF_DYNAMIC),
     SDEI_PRIVATE_EVENT(101, SDEI_DYN_IRQ, SDEI_MAPF_DYNAMIC)

     /* Events for explicit dispatch */
     SDEI_EXPLICIT_EVENT(2000, SDEI_MAPF_NORMAL);
     SDEI_EXPLICIT_EVENT(2000, SDEI_MAPF_CRITICAL);
};

/* Shared event mappings */
static sdei_ev_map_t plat_shared_sdei[] = {
     SDEI_SHARED_EVENT(804, 0, SDEI_MAPF_DYNAMIC),

     /* Dynamic shared events */
     SDEI_SHARED_EVENT(3000, SDEI_DYN_IRQ, SDEI_MAPF_DYNAMIC),
     SDEI_SHARED_EVENT(3001, SDEI_DYN_IRQ, SDEI_MAPF_DYNAMIC)
};

/* Export SDEI events */
REGISTER_SDEI_MAP(plat_private_sdei, plat_shared_sdei);
```

4.11.4 Configuration within Exception Handling Framework

The SDEI dispatcher functions alongside the Exception Handling Framework. This means that the platform must assign priorities to both Normal and Critical SDEI interrupts for the platform:

- Install priority descriptors for Normal and Critical SDEI interrupts.
- For those interrupts that are statically bound (i.e. events defined as having the SDEI_MAPF_BOUND property), enumerate their properties for the GIC driver to configure interrupts accordingly.

  The interrupts must be configured to target EL3. This means that they should be configured as *Group 0*. Additionally, on GICv2 systems, the build option GICV2_G0_FOR_EL3 must be set to 1.

See also *SDEI porting requirements*.

4.11.5 Determining client EL

The SDEI specification requires that the *physical* SDEI client executes in the highest Non-secure EL implemented on the system. This means that the dispatcher will only allow SDEI calls to be made from:

- EL2, if EL2 is implemented. The Hypervisor is expected to implement a *virtual* SDEI dispatcher to support SDEI clients in Guest Operating Systems executing in Non-secure EL1.
- Non-secure EL1, if EL2 is not implemented or disabled.

See the function `sdei_client_el()` in `sdei_private.h`.

4.11.6 Explicit dispatch of events

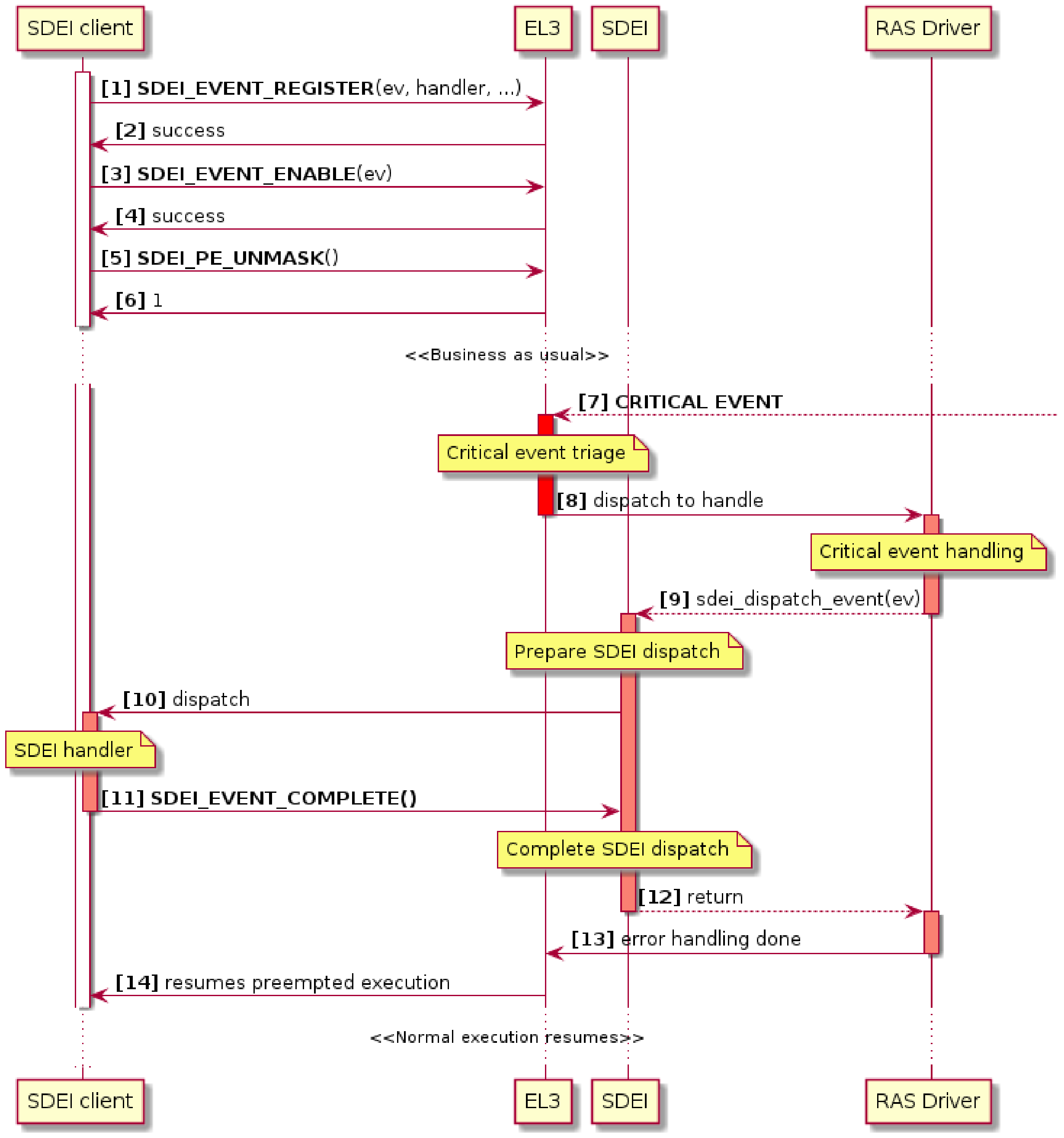
Typically, an SDEI event dispatch is caused by the PE receiving interrupts that are bound to an SDEI event. However, there are cases where the Secure world requires dispatch of an SDEI event as a direct or indirect result of a past activity, such as receiving a Secure interrupt or an exception.

The SDEI dispatcher implementation provides `sdei_dispatch_event()` API for this purpose. The API has the following signature:

```
int sdei_dispatch_event(int ev_num);
```

The parameter `ev_num` is the event number to dispatch. The API returns `0` on success, or `-1` on failure.

The following figure depicts a scenario involving explicit dispatch of SDEI event. A commentary is provided below:



As part of initialisation, the SDEI client registers a handler for a platform event [1], enables the event [3], and unmasks the current PE [5]. Note that, unlike in *general SDEI dispatch*, this doesn't involve interrupt binding, as bound or dynamic events can't be explicitly dispatched (see the section below).

At a later point in time, a critical event[2] is trapped into EL3 [7]. EL3 performs a first-level triage of the event, and a RAS component assumes further handling [8]. The dispatch completes, but intends to involve Non-secure world in further handling, and therefore decides to explicitly dispatch an event [10] (which the client had already registered for [1]). The rest of the sequence is similar to that in the *general SDEI dispatch*: the requested event is dispatched to the client (assuming all the conditions are met), and when the handler completes, the preempted execution resumes.

[2] Examples of critical events are *SError*, *Synchronous External Abort*, *Fault Handling interrupt* or *Error Recovery interrupt* from one of RAS nodes in the system.

Conditions for event dispatch

All of the following requirements must be met for the API to return `0` and event to be dispatched:

- SDEI events must be unmasked on the PE. I.e. the client must have called `PE_UNMASK` beforehand.
- Event 0 can't be dispatched.
- The event must be declared using the `SDEI_EXPLICIT_EVENT()` macro described above.
- The event must be private to the PE.
- The event must have been registered for and enabled.
- A dispatch for the same event must not be outstanding. I.e. it hasn't already been dispatched and is yet to be completed.
- The priority of the event (either Critical or Normal, as configured by the platform at build-time) shouldn't cause priority inversion. This means:
  - If it's of Normal priority, neither Normal nor Critical priority dispatch must be outstanding on the PE.
  - If it's of a Critical priority, no Critical priority dispatch must be outstanding on the PE.

Further, the caller should be aware of the following assumptions made by the dispatcher:

- The caller of the API is a component running in EL3; for example, a RAS driver.
- The requested dispatch will be permitted by the Exception Handling Framework. I.e. the caller must make sure that the requested dispatch has sufficient priority so as not to cause priority level inversion within Exception Handling Framework.
- The caller must be prepared for the SDEI dispatcher to restore the Non-secure context, and mark that the active context.
- The call will block until the SDEI client completes the event (i.e. when the client calls either `SDEI_EVENT_COMPLETE` or `SDEI_COMPLETE_AND_RESUME`).
- The caller must be prepared for this API to return failure and handle accordingly.

4.11.7 Porting requirements

The porting requirements of the SDEI dispatcher are outlined in the *Porting Guide*.

4.11.8 Note on writing SDEI event handlers

This section pertains to SDEI event handlers in general, not just when using the TF-A SDEI dispatcher.

The SDEI specification requires that event handlers preserve the contents of all registers except `x0` to `x17`. This has significance if event handler is written in C: compilers typically adjust the stack frame at the beginning and end of C functions. For example, AArch64 GCC typically produces the following function prologue and epilogue:

```
c_event_handler:
        stp     x29, x30, [sp,#-32]!
        mov     x29, sp

        ...

        bl      ...
```

(continues on next page)

(continued from previous page)

```
        ...

        ldp     x29, x30, [sp],#32
        ret
```

The register `x29` is used as frame pointer in the prologue. Because neither a valid `SDEI_EVENT_COMPLETE` nor `SDEI_EVENT_COMPLETE_AND_RESUME` calls return to the handler, the epilogue never gets executed, and registers `x29` and `x30` (in the case above) are inadvertently corrupted. This violates the SDEI specification, and the normal execution thereafter will result in unexpected behaviour.

To work this around, it's advised that the top-level event handlers are implemented in assembly, following a similar pattern as below:

```
asm_event_handler:
        /* Save link register whilst maintaining stack alignment */
        stp     xzr, x30, [sp, #-16]!
        bl      c_event_handler

        /* Restore link register */
        ldp     xzr, x30, [sp], #16

        /* Complete call */
        ldr     x0, =SDEI_EVENT_COMPLETE
        smc     #0
        b       .
```

4.12 Secure Partition Manager

Contents

4.12.1 Acronyms

DTB	Device Tree Blob
DTS	Device Tree Source
EC	Execution Context
FIP	Firmware Image Package
FF-A	Firmware Framework for A-class
IPA	Intermediate Physical Address
NWd	Normal World
ODM	Original Design Manufacturer
OEM	Original Equipment Manufacturer
PA	Physical Address
PE	Processing Element
PVM	Primary VM
PSA	Platform Security Architecture
SP	Secure Partition
SPM	Secure Partition Manager
SPMC	SPM Core
SPMD	SPM Dispatcher
SiP	Silicon Provider
SWd	Secure World
TLV	Tag-Length-Value
TOS	Trusted Operating System
VM	Virtual Machine

4.12.2 Foreword

Two implementations of a Secure Partition Manager co-exist in the TF-A codebase:

- SPM based on the PSA FF-A specification *[1]*.
- SPM based on the MM interface to communicate with an S-EL0 partition *[2]*.

Both implementations differ in their architectures and only one can be selected at build time.

This document:

- describes the PSA FF-A implementation where the Secure Partition Manager resides at EL3 and S-EL2 (or EL3 and S-EL1).
- is not an architecture specification and it might provide assumptions on sections mandated as implementation-defined in the specification.
- covers the implications to TF-A used as a bootloader, and Hafnium used as a reference code base for an S-EL2 secure firmware on platforms implementing Armv8.4-SecEL2.

Terminology

- Hypervisor refers to the NS-EL2 component managing Virtual Machines (or partitions) in the Normal World.
- SPMC refers to the S-EL2 component managing Virtual Machines (or Secure Partitions) in the Secure World when Armv8.4-SecEL2 extension is implemented.
- Alternatively, SPMC can refer to an S-EL1 component, itself being a Secure Partition and implementing the FF-A ABI on pre-Armv8.4 platforms.
- VM refers to a Normal World Virtual Machine managed by an Hypervisor.
- SP refers to a Secure World "Virtual Machine" managed by the SPMC component.

Support for legacy platforms

In the implementation, the SPM is split into SPMD and SPMC components (although not strictly mandated by the specification). SPMD is located at EL3 and principally relays FF-A messages from NWd (Hypervisor or OS kernel) to SPMC located either at S-EL1 or S-EL2.

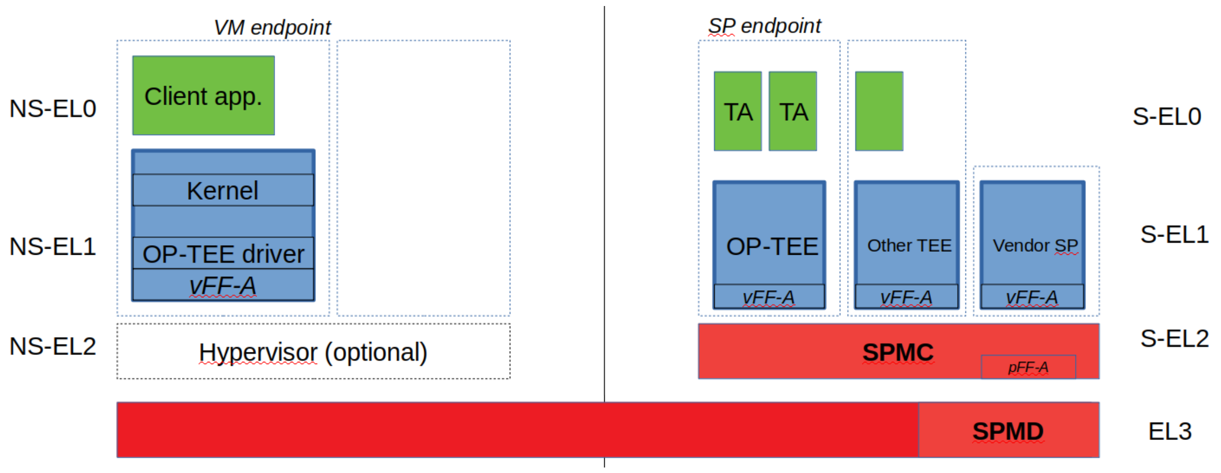
Hence TF-A must support both cases where SPMC is either located at:

- S-EL1 supporting pre-Armv8.4 platforms. SPMD conveys FF-A protocol from EL3 to S-EL1.
- S-EL2 supporting platforms implementing Armv8.4-SecEL2 extension. SPMD conveys FF-A protocol from EL3 to S-EL2.

The same SPMD component is used to support both configurations. The SPMC execution level is a build time choice.

4.12.3 Sample reference stack

The following diagram illustrates a possible configuration with SPMD and SPMC, one or multiple Secure Partitions, with or without an optional Hypervisor:



4.12.4 TF-A build options

The following TF-A build options are provisioned:

- **SPD=spmd**: this option selects the SPMD component to relay FF-A protocol from NWd to SWd back and forth. It is not possible to enable another Secure Payload Dispatcher when this option is chosen.
- **SPMD_SPM_AT_SEL2**: this option adjusts the SPMC execution level to being S-EL1 or S-EL2. It defaults to enabled (value 1) when SPD=spmd is chosen.
- **CTX_INCLUDE_EL2_REGS**: this option permits saving (resp. restoring) the EL2 system register context before entering (resp. after leaving) the SPMC. It is mandatory when `SPMD_SPM_AT_SEL2` is enabled. The context save/restore routine and exhaustive list of registers is visible at *[4]*.
- **SP_LAYOUT_FILE**: this option provides a text description file providing paths to SP binary images and DTS format manifests (see *Specifying partition binary image and DT*). It is required when `SPMD_SPM_AT_SEL2` is enabled hence when multiple secure partitions are to be loaded on behalf of SPMC.

	CTX_INCLUDE_EL2_REGS	SPMD_SPM_AT_SEL2
SPMC at S-EL1 (e.g. OP-TEE)	0	0
SPMC at S-EL2 (e.g. Hafnium)	1	**1 (default when** SPD=spmd)

Other combinations of such build options either break the build or are not supported.

Note, the `CTX_INCLUDE_EL2_REGS` option provides the generic support for barely saving/restoring EL2 registers from an Arm arch perspective. As such it is decoupled from the `SPD=spmd` option.

BL32 option is re-purposed to specify the SPMC image. It can specify either the Hafnium binary path (built for the secure world) or the path to a TEE binary implementing the FF-A protocol.

BL33 option can specify either:

- the TFTF binary or
- the Hafnium binary path (built for the normal world) if VMs were loaded by TF-A beforehand or
- a minimal loader performing the loading of VMs and Hafnium.

Sample TF-A build command line when SPMC is located at S-EL1 (typically pre-Armv8.4):

```
make \
CROSS_COMPILE=aarch64-none-elf- \
SPD=spmd \
SPMD_SPM_AT_SEL2=0 \
BL32=<path-to-tee-binary> \
BL33=<path-to-nwd-binary> \
PLAT=fvp \
all fip
```

Sample TF-A build command line for an Armv8.4-SecEL2 enabled system where SPMC is located at S-EL2:

```
make \
CROSS_COMPILE=aarch64-none-elf- \
SPD=spmd \
CTX_INCLUDE_EL2_REGS=1 \
ARM_ARCH_MINOR=4 \
BL32=<path-to-swd-hafnium-binary>
BL33=<path-to-nwd-binary> \
```

(continues on next page)

(continued from previous page)

```
SP_LAYOUT_FILE=sp_layout.json \
PLAT=fvp \
all fip
```

Build options to enable secure boot:

```
make \
CROSS_COMPILE=aarch64-none-elf- \
SPD=spmd \
CTX_INCLUDE_EL2_REGS=1 \
ARM_ARCH_MINOR=4 \
BL32=<path-to-swd-hafnium-binary>
BL33=<path-to-nwd-binary> \
SP_LAYOUT_FILE=../tf-a-tests/build/fvp/debug/sp_layout.json \
MBEDTLS_DIR=<path-to-mbedtls-lib> \
TRUSTED_BOARD_BOOT=1 \
COT=dualroot \
ARM_ROTPK_LOCATION=devel_rsa \
ROT_KEY=plat/arm/board/common/rotpk/arm_rotprivk_rsa.pem \
GENERATE_COT=1 \
PLAT=fvp \
all fip
```

4.12.5 Boot process

Loading Hafnium and Secure Partitions in the secure world

The Hafnium implementation in normal world requires VMs to be loaded in memory prior to booting. The mechanism upon which VMs are loaded and exposed to Hafnium are either:

- by supplying a ramdisk image where VM images are concatenated (1)
- or by providing VM load addresses within Hafnium manifest (2)

TF-A is the bootlader for the Hafnium and SPs in the secure world. TF-A does not provide tooling or libraries manipulating ramdisks as required by (1). Thus BL2 loads SPs payloads independently. SPs may be signed by different parties (SiP, OEM/ODM, TOS vendor, etc.). Thus they are supplied as distinct “self-contained” signed entities within the FIP flash image. The FIP image itself is not signed hence providing ability to upgrade SPs in the field.

Booting through TF-A

SP manifests

An SP manifest describes SP attributes as defined in *[1]* section 3.1 (partition manifest at virtual FF-A instance) in DTS text format. It is represented as a single file associated with the SP. A sample is provided by *[5]*. A binding document is provided by *[6]*.

Secure Partition packages

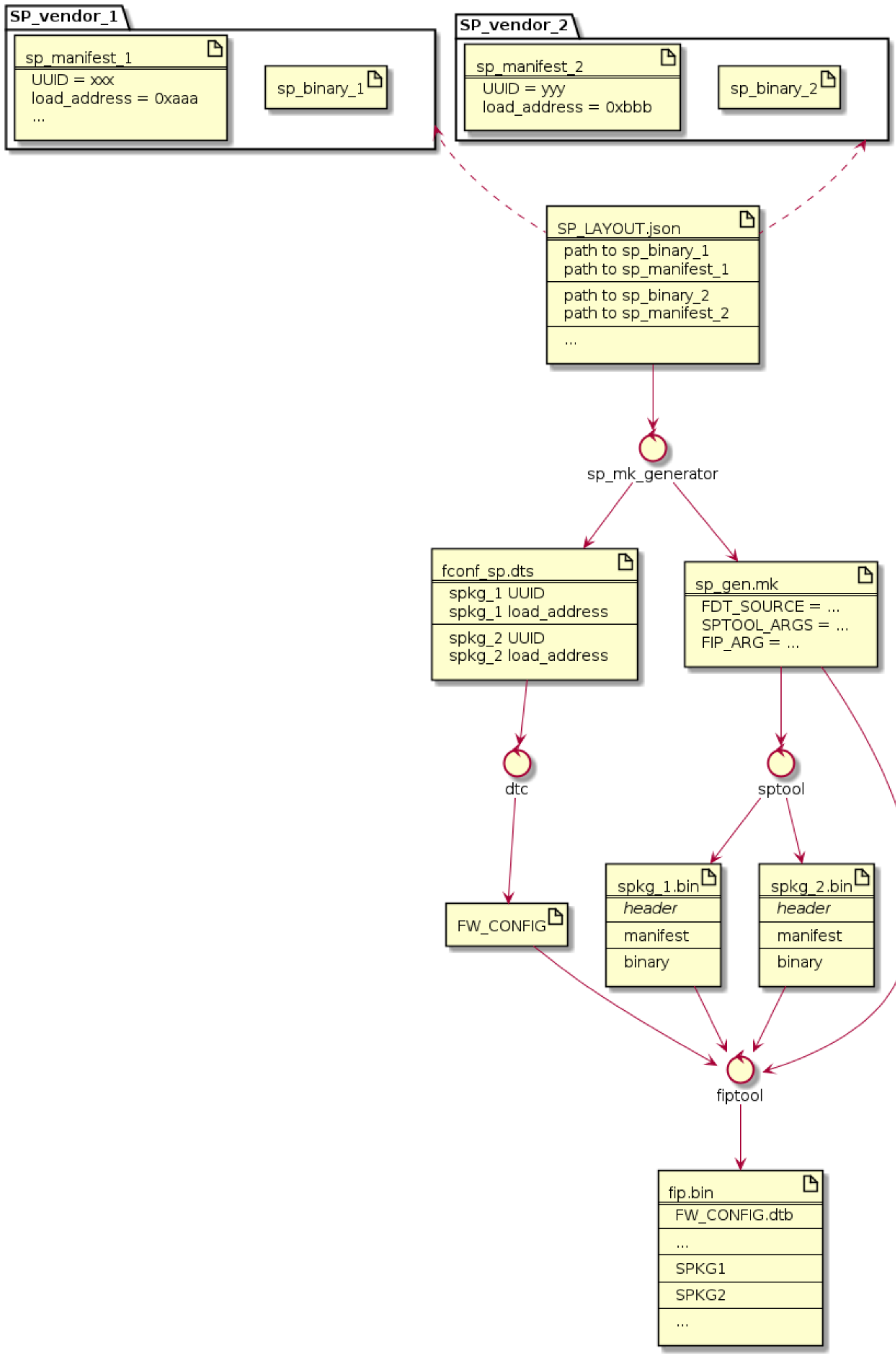
Secure Partitions are bundled as independent package files consisting of:

- a header
- a DTB
- an image payload

The header starts with a magic value and offset values to SP DTB and image payload. Each SP package is loaded independently by BL2 loader and verified for authenticity and integrity.

The SP package identified by its UUID (matching FF-A uuid) is inserted as a single entry into the FIP at end of the TF-A build flow as shown:

```
Trusted Boot Firmware BL2: offset=0x1F0, size=0x8AE1, cmdline="--tb-fw"
EL3 Runtime Firmware BL31: offset=0x8CD1, size=0x13000, cmdline="--soc-fw"
Secure Payload BL32 (Trusted OS): offset=0x1BCD1, size=0x15270, cmdline="--tos-fw"
Non-Trusted Firmware BL33: offset=0x30F41, size=0x92E0, cmdline="--nt-fw"
HW_CONFIG: offset=0x3A221, size=0x2348, cmdline="--hw-config"
TB_FW_CONFIG: offset=0x3C569, size=0x37A, cmdline="--tb-fw-config"
SOC_FW_CONFIG: offset=0x3C8E3, size=0x48, cmdline="--soc-fw-config"
TOS_FW_CONFIG: offset=0x3C92B, size=0x427, cmdline="--tos-fw-config"
NT_FW_CONFIG: offset=0x3CD52, size=0x48, cmdline="--nt-fw-config"
B4B5671E-4A90-4FE1-B81F-FB13DAE1DACB: offset=0x3CD9A, size=0xC168, cmdline="--blob"
D1582309-F023-47B9-827C-4464F5578FC8: offset=0x48F02, size=0xC168, cmdline="--blob"
```


SP_vendor_1
sp_manifest_1
UUID = xxx
load_address = 0xaaa
...
sp_binary_1
SP_vendor_2
sp_manifest_2
UUID = yyy
load_address = 0xbbb
sp_binary_2
SP_LAYOUT.json
path to sp_binary_1
path to sp_manifest_1
path to sp_binary_2
path to sp_manifest_2
...
sp_mk_generator
fconf_sp.dts
spkg_1 UUID
spkg_1 load_address
spkg_2 UUID
spkg_2 load_address
sp_gen.mk
FDT_SOURCE = ...
SPTOOL_ARGS = ...
FIP_ARG = ...
dtc
sptool
FW_CONFIG
spkg_1.bin
header
manifest
binary
spkg_2.bin
header
manifest
binary
fiptool
fip.bin
FW_CONFIG.dtb
...
SPKG1
SPKG2
...

Specifying partition binary image and DT

A description file (json format) is passed to the build flow specifying paths to the SP binary image and associated DTS partition manifest file. The latter is going through the dtc compiler to generate the dtb fed into the SP package. This file also specifies the owner of the SP, which is an optional field and identifies the signing domain in case of dualroot CoT. The possible owner of an SP could either be Silicon Provider or Platform, and the corresponding "owner" field value could either be "SiP" or "Plat". In absence of "owner" field, it defaults to "SiP".

```
{
    "tee1" : {
        "image": "tee1.bin",
         "pm": "tee1.dts",
         "owner": "SiP"
    },

    "tee2" : {
        "image": "tee2.bin",
        "pm": "tee2.dts",
        "owner": "Plat"
    }
}
```

SPMC manifest

This manifest contains an SPMC attributes node consumed by SPMD at boot time. It is implementing the description from *[1]* section 3.2 (SP manifest at physical FF-A instance). The SP manifest at physical FF-A instance is used by the SPMD to setup a SP that co-resides with the SPMC and executes at S-EL1 or Secure Supervisor mode.

In this implementation its usage is extended to the secure physical FF-A instance where SPMC executes at S-EL2.

```
attribute {
    spmc_id = <0x8000>;
    maj_ver = <0x1>;
    min_ver = <0x0>;
    exec_state = <0x0>;
    load_address = <0x0 0x6000000>;
    entrypoint = <0x0 0x6000000>;
    binary_size = <0x60000>;
};
```

- *spmc_id* defines the endpoint ID value that SPMC can query through `FFA_ID_GET`.
- *maj_ver/min_ver*. SPMD checks provided version versus its internal version and aborts if not matching.
- *exec_state* defines SPMC execution state (can be AArch64 for Hafnium, or AArch64/AArch32 for OP-TEE at S-EL1).
- *load_address* and *binary_size* are mostly used to verify secondary entry points fit into the loaded binary image.
- *entrypoint* defines the cold boot primary core entry point used by SPMD (currently matches `BL32_BASE`)

Other nodes in the manifest are consumed by Hafnium in the secure world. A sample can be found at [7]:

- The *chosen* node is currently unused in SWd. It is meant for NWd to specify the init ramdisk image.
- The *hypervisor* node describes SPs. *is_ffa_partition* boolean attribute indicates an SP. Load-addr field specifies the load address at which TF-A loaded the SP package.

- *cpus* node provide the platform topology and allows MPIDR to VMPIDR mapping. Notice with current implementation primary cpu is declared first, then secondary cpus must be declared in reverse order.

SPMC boot

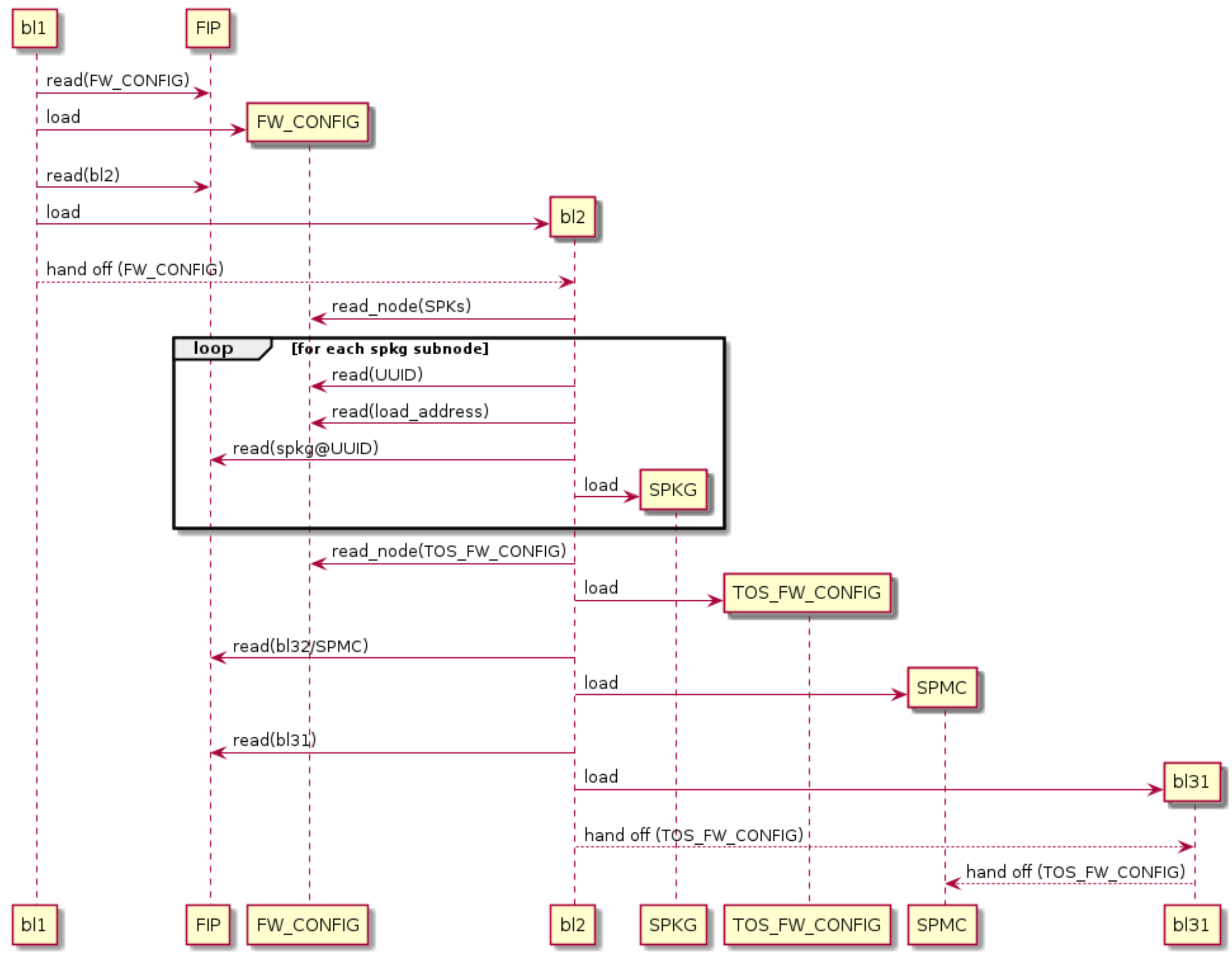
The SPMC is loaded by BL2 as the BL32 image.

The SPMC manifest is loaded by BL2 as the `TOS_FW_CONFIG` image.

BL2 passes the SPMC manifest address to BL31 through a register.

BL31(SPMD) runs from primary core, initializes the core contexts and launches BL32 passing the SPMC manifest address through a register.

Loading of SPs



Notice this boot flow is an implementation sample on Arm's FVP platform. Platforms not using FW_CONFIG would adjust to a different implementation.

Secure boot

The SP content certificate is inserted as a separate FIP item so that BL2 loads SPMC, SPMC manifest and Secure Partitions and verifies them for authenticity and integrity. Refer to TBBR specification *[3]*.

The multiple-signing domain feature (in current state dual signing domain) allows the use of two root keys namely S-ROTPK and NS-ROTPK (see *[8]*):

- SPMC (BL32) and SPMC manifest are signed by the SiP using the S-ROTPK.
- BL33 may be signed by the OEM using NS-ROTPK.
- An SP may be signed either by SiP (using S-ROTPK) or by OEM (using NS-ROTPK).

Longer term multiple signing domain will allow additional signing keys, e.g. if SPs originate from different parties.

See *TF-A build options* for a sample build command line.

4.12.6 Hafnium in the secure world

NOTE: this section is work in progress. Descriptions and implementation choices are subject to evolve.

General considerations

Build platform for the secure world

The implementation might add specific code parts only relevant to the secure world. Such code parts might be isolated into different files and/or conditional code enclosed by a `SECURE_WORLD` macro.

Secure Partitions CPU scheduling

In the normal world, VMs are scheduled by the FFA_RUN ABI invoked from the primary scheduler (in the primary VM), or by a direct message request or response.

With the FF-A EAC specification, Secure Partitions are scheduled by direct message invocations from a NWd VM or another SP.

Platform topology

As stated in *[1]* section 4.4.1 the SPMC implementation assumes the following SP types:

- Pinned MP SPs: an Execution Context id matches a physical PE id. MP SPs must implement the same number of ECs as the number of PEs in the platform. Hence the *execution-ctx-count* as defined by *[1]* (or NWd-Hafnium *vcpu_count*) can only take the value of one or the number of physical PEs.
- Migratable UP SPs: a single execution context can run and be migrated on any physical PE. It declares a single EC in its SP manifest. An UP SP can receive a direct message request on any physical core.

Usage of PSCI services in the secure world

- The normal world Hypervisor (optional) or OS kernel issues PSCI service invocations e.g. to request PSCI version, wake-up a secondary core, or request core suspend. This happens at the non-secure physical FF-A instance. In the example case of Hafnium in the normal world, it boots on the primary core and one of the first initialization step is to request the PSCI version. It then launches the primary VM. The primary VM upon initializing performs PSCI service calls (at non-secure virtual FF-A instance) which are trapped by the Hypervisor. Invocation from OS Kernel ends straight at EL3. The PVM issues `PSCI_CPU_ON` service calls to wake-up secondary cores by passing an `MPIDR`, entry point address and a CPU context address. The EL3 PSCI layer then performs an exception return to the secondary core entry point on the targeted core. Other PSCI calls can happen at run-time from the PVM e.g. to request core suspend.
- In the existing TF-A PSCI standard library, PSCI service calls are filtered at EL3 to only originate from the NWd. Thus concerning the SPMC (at secure physical FF-A instance) the PSCI service invocations cannot happen as in the normal world. For example, a `PSCI_CPU_ON` service invocation from the SPMC does not reach the PSCI layer.

Parsing SP partition manifests

Hafnium must be able to consume SP manifests as defined in *[1]* section 3.1, at least for the mandatory fields.

The SP manifest may contain memory and device regions nodes.

- Memory regions shall be mapped in the SP Stage-2 translation regime at load time. A memory region node can specify RX/TX buffer regions in which case it is not necessary for an SP to explicitly call the `FFA_RXTX_MAP` service.
- Device regions shall be mapped in SP Stage-2 translation regime as peripherals and possibly allocate additional resources (e.g. interrupts)

Base addresses for memory and device region nodes are IPAs provided SPMC identity maps IPAs to PAs within SP Stage-2 translation regime.

Note: currently both VTTBR_EL2 and VSTTBR_EL2 resolve to the same set of page tables. It is still open whether two sets of page tables shall be provided per SP. The memory region node as defined in the spec (section 3.1 Table 10) provides a memory security attribute hinting to map either to the secure or non-secure stage-2 table.

Passing boot data to the SP

[1] Section 3.4.2 "Protocol for passing data" defines a method to passing boot data to SPs (not currently implemented).

Provided that the whole Secure Partition package image (see *Secure Partition packages*) is mapped to the SP's secure Stage-2 translation regime, an SP can access its own manifest DTB blob and extract its partition manifest properties.

SP Boot order

SP manifests provide an optional boot order attribute meant to resolve dependencies such as an SP providing a service required to properly boot another SP.

Boot phases

Primary core boot-up

The SPMC performs its platform initializations then loads and creates secure partitions based on SP packages and manifests. Then each secure partition is launched in sequence (see *SP Boot order*) on their primary Execution Context.

Notice the primary physical core may not be core 0. Hence if the primary core linear id is N, the 1:1 mapping requires MP SPs are launched using EC[N] on PE[N] (see *Platform topology*).

The SP's primary Execution Context (or the EC used when the partition is booted) exits through `FFA_MSG_WAIT` to indicate successful initialization.

Secondary physical core boot-up

Upon boot-up, the SPMC running on the primary core performs implementation-defined SPMD service calls at secure physical FF-A instance to register the secondary physical cores entry points and context information:

- This is done through a direct message request invocation to the SPMD (`SET_ENTRY_POINT`). This service call does not wake-up the targeted core immediately. The secondary core is woken up later by a NWd `PSCI_CPU_ON` service invocation. A notification is passed from EL3 PSCI layer to the SPMD, and then to SPMC through an implementation-defined interface.
- The SPMC/SPMD interface can consist of FF-A direct message requests/responses transporting PM events.

If there is no Hypervisor in the normal world, the OS Kernel issues `PSCI_CPU_ON` calls that are directly trapped to EL3.

When a secondary physical core wakes-up the SPMD notifies the SPMC which updates its internal states reflecting current physical core is being turned on. It might then return straight to the SPMD and then to the NWd.

(under discussion) There may be possibility that an SP registers "PM events" (during primary EC boot stage) through an ad-hoc interface. Such events would be relayed by SPMC to one or more registered SPs on need basis (see *Power management*).

Secondary virtual core boot-up

In the example case where Hafnium exists in the normal world, secondary VMs issue a `PSCI_CPU_ON` service call which is trapped to the Hypervisor. The latter then enables the vCPU context for the targeted core, and switches to the PVM down to the kernel driver with an `HF_WAKE_UP` message. The NWd driver in PVM can then schedule the newly woken up vCPU context.

In the secure world the primary EC of a given SP passes the secondary EC entry point and context. The SMC service call is trapped into the SPMC. This can be either *(under discussion)*:

- a specific interface registering the secondary EC entry point, similarly to above `SET_ENTRY_POINT` service.
- Re-purposing the `PSCI_CPU_ON` function id. It is assumed that even if the input arguments are the same as the ones defined in the PSCI standard, the usage deviates by the fact the secondary EC is not woken up immediately. At least for the PSA-FF-A EAC where only direct messaging is allowed, it is only after the first direct message invocation that the secondary EC is entered. This option might be preferred when the same code base is re-used for a VM or an SP. The ABI to wake-up a secondary EC can remain similar.

SPs are always scheduled from the NWd, this paradigm did not change from legacy TEEs. There must always be some logic (or driver) in the NWd to relinquish CPU cycles to the SWd. If primary core is 0, an SP EC[x>0] entry point is supplied by the SP EC[0] when the system boots in SWd. But this EC[x] is not immediately entered at boot. Later in the boot process when NWd is up, a direct message request issued from physical core 1 ends up in SP EC[1], and only at this stage this context is effectively scheduled.

It should be possible for an SP to call into another SP through direct message provided the latter SP has been booted already. The "boot-order" field in partition manifests (*SP Boot order*) fulfills the dependency towards availability of a service within an SP offered to another SP.

Mandatory interfaces

The following interfaces must be exposed to any VM or SP:

- `FFA_STATUS`
- `FFA_ERROR`
- `FFA_INTERRUPT`
- `FFA_VERSION`
- `FFA_FEATURES`
- `FFA_RX_RELEASE`
- `FFA_RXTX_MAP`
- `FFA_RXTX_UNMAP`
- `FFA_PARTITION_INFO_GET`
- `FFA_ID_GET`

FFA_VERSION

Per *[1]* section 8.1 `FFA_VERSION` requires a *requested_version* parameter from the caller.

In the current implementation when `FFA_VERSION` is invoked from:

- Hypervisor in NS-EL2: the SPMD returns the SPMC version specified in the SPMC manifest.
- OS kernel in NS-EL1 when NS-EL2 is not present: the SPMD returns the SPMC version specified in the SPMC manifest.
- VM in NWd: the Hypervisor returns its implemented version.
- SP in SWd: the SPMC returns its implemented version.
- SPMC at S-EL1/S-EL2: the SPMD returns its implemented version.

FFA_FEATURES

FF-A features may be discovered by Secure Partitions while booting through the SPMC. However, SPMC cannot get features from Hypervisor early at boot time as NS world is not setup yet.

The Hypervisor may decide to gather FF-A features from SPMC through SPMD once at boot time and store the result. Later when a VM requests FF-A features, the Hypervisor can adjust its own set of features with what SPMC advertised, if necessary. Another approach is to always forward FF-A features to the SPMC when a VM requests it to the Hypervisor. Although the result is not supposed to change over time so there may not be added value doing the systematic forwarding.

FFA_RXTX_MAP/FFA_RXTX_UNMAP

VM mailboxes are re-purposed to serve as SP RX/TX buffers. The RX/TX map API maps the send and receive buffer IPAs to the SP Stage-2 translation regime.

Hafnium in the normal world defines VMs and their attributes as logical structures, including a mailbox used for FF-A indirect messaging, memory sharing, or the *FFA_PARTITION_INFO_GET* ABI. This same mailbox structure is re-used in the SPMC. *[1]* states only direct messaging is allowed to SPs. Thus mailbox usage is restricted to implementing *FFA_PARTITION_INFO_GET* and memory sharing ABIs.

FFA_PARTITION_INFO_GET

Partition info get service call can originate:

- from SP to SPM
- from VM to Hypervisor
- from Hypervisor to SPM

For the latter case, the service call must be forwarded through the SPMD.

FFA_ID_GET

The SPMD returns:

- a default zero value on invocation from the Hypervisor.
- The `spmc_id` value specified in the SPMC manifest on invocation from the SPMC (see *SPMC manifest*)

The FF-A id space is split into a non-secure space and secure space:

- FF-A id with bit 15 clear refer to normal world VMs.
- FF-A id with bit 15 set refer to secure world SPs

Such convention helps the SPMC discriminating the origin and destination worlds in an FF-A service invocation. In particular the SPMC shall filter unauthorized transactions in its world switch routine. It must not be permitted for a VM to use a secure FF-A id as origin world through spoofing:

- A VM-to-SP messaging passing shall have an origin world being non-secure (FF-A id bit 15 clear) and destination world being secure (FF-A id bit 15 set).
- Similarly, an SP-to-SP message shall have FF-A id bit 15 set for both origin and destination ids.

An incoming direct message request arriving at SPMD from NWd is forwarded to SPMC without a specific check. The SPMC is resumed through eret and “knows” the message is coming from normal world in this specific code path. Thus the origin endpoint id must be checked by SPMC for being a normal world id.

An SP sending a direct message request must have bit 15 set in its origin endpoint id and this can be checked by the SPMC when the SP invokes the ABI.

The SPMC shall reject the direct message if the claimed world in origin endpoint id is not consistent:

- It is either forwarded by SPMD and thus origin endpoint id must be a “normal world id”,
- or initiated by an SP and thus origin endpoint id must be a “secure world id”.

Direct messaging

This is a mandatory interface for Secure Partitions consisting in direct message request and responses.

The `ffa_handler` Hafnium function may:

- trigger a world change e.g. when an SP invokes the direct message response ABI to a VM.
- handle multiple requests from the NWd without resuming an SP.

SP-to-SP

- An SP can send a direct message request to another SP
- An SP can receive a direct message response from another SP.

VM-to-SP

- A VM can send a direct message request to an SP
- An SP can send a direct message response to a VM

SPMC-SPMD messaging

Specific implementation-defined endpoint IDs are allocated to the SPMC and SPMD. Referring those IDs in source/destination fields of a direct message request/response permits SPMD to SPMC messaging back and forth.

Per *[1]* Table 114 Config No. 1 (physical FF-A instance):

- SPMC=>SPMD direct message request uses SMC conduit
- SPMD=>SPMC direct message request uses ERET conduit

Per *[1]* Table 118 Config No. 1 (physical FF-A instance):

- SPMC=>SPMD direct message response uses SMC conduit
- SPMD=>SPMC direct message response uses ERET conduit

Memory management

This section only deals with the PE MMU configuration.

Hafnium in the normal world deals with NS buffers only and provisions a single root page table directory to VMs. In context of S-EL2 enabled firmware, two IPA spaces are output from Stage-1 translation (secure and non-secure). The Stage-2 translation handles:

- A single secure IPA space when an SP Stage-1 MMU is disabled.
- Two IPA spaces (secure and non-secure) when Stage-1 MMU is enabled.

`VTCR_EL2` and `VSTCR_EL2` provide additional bits for controlling the NS/S IPA translations (`VSTCR_EL2.SW`, `VSTCR_EL2.SA`, `VTCR_EL2.NSW`, `VTCR_EL2.NSA`). There may be two approaches:

- secure and non-secure mappings are rooted as two separate root page tables
- secure and non-secure mappings use the same root page table. Access from S-EL1 to an NS region translates to a secure physical address space access.

Interrupt management

Road to a para-virtualized interface

Current Hafnium implementation uses an ad-hoc mechanism for a VM to get a pending interrupt number through an hypercall. The PVM injects interrupts to VMs by delegation from the Hypervisor. The PVM probes a pending interrupt directly from the GIC distributor.

The short-term plan is to have Hafnium/SPMC in the secure world owner of the GIC configuration.

The SPMC fully owns the GIC configuration at S-EL2. The SPMC manages interrupt resources and allocates interrupt ID based on SP manifests. The SPMC acknowledges physical interrupts and injects virtual interrupts by setting the vIRQ bit when resuming an SP. A Secure Partition gathers the interrupt number through an hypercall.

Notice the SPMC/SPMD has to handle Group0 secure interrupts in addition to Group1 S/NS interrupts.

Power management

Assumption on the Nwd:

- NWd is the best candidate to own the platform Power Management policy. It is master to invoking PSCI service calls from physical CPUs.
- EL3 monitor is in charge of the PM control part (its PSCI layer actually writing to platform registers).
- It is fine for the Hypervisor to trap PSCI calls and relay to EL3, or OS kernel driver to emit PSCI service calls.

PSCI notification are relayed through the SPMD/SPD PM hooks to the SPMC. This can either be through re-use of PSCI FIDs or an FF-A direct message from SPMD to SPMC.

The SPMD performs an exception return to the SPMC which is resumed to its `eret_handler` routine. It is then either consuming a PSCI FID or an FF-A FID. Depending on the servicing, the SPMC may return directly to the SPMD (and then NWd) without resuming an SP at this stage. An example of this is invocation of `FFA_PARTITION_INFO_GET` from NWd relayed by the SPMD to the SPMC. The SPMC returns the needed partition information to the SPMD (then NWd) without actually resuming a partition in secure world.

(under discussion) About using PSCI FIDs from SPMD to SPMC to notify of PM events, it is still questioned what to use as the return code from the SPMC. If the function ID used by the SPMC is not an FF-A ID when doing SMC, then the EL3 std svc handler won't route the response to the SPMD. That's where comes the idea to embed the notification into an FF-A message. The SPMC can discriminate this message as being a PSCI event, process it, and reply with an FF-A return message that the SPMD receives as an acknowledgement.

SP notification

Power management notifications are conveyed from PSCI library to the SPMD / SPD hooks. A range of events can be relayed to SPMC.

SPs may need to be notified about specific PM events.

- SPs might register PM events to the SPMC
- On SPMD to SPMC notification, a limited range of SPs may be notified through a direct message.
- This assumes the mentioned SPs supports managed exit.

The SPMC is the first to be notified about PM events from the SPMD. It is up to the SPMC to arbitrate to which SP it needs to send PM events. An SP explicitly registers to receive notifications to specific PM events. The register operation can either be an implementation-defined service call to the SPMC when the primary SP EC boots, or be supplied through the SP manifest.

4.12.7 References

[1] Platform Security Architecture Firmware Framework for Arm® v8-A 1.0 Platform Design Document

[2] *Secure Partition Manager using MM interface*

[3] Trusted Boot Board Requirements Client

[4] https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/tree/lib/el3_runtime/aarch64/context.S#n45

[5] https://git.trustedfirmware.org/TF-A/tf-a-tests.git/tree/spm/cactus/cactus.dts

[6] https://trustedfirmware-a.readthedocs.io/en/latest/components/psa-ffa-manifest-binding.html

[7] https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/tree/plat/arm/board/fvp/fdts/fvp_spmc_manifest.dts

[8] https://developer.trustedfirmware.org/w/tf_a/poc-multiple-signing-domains/

4.13 Secure Partition Manager (MM)

4.13.1 Foreword

Two implementations of a Secure Partition Manager co-exist in the TF-A codebase:

- SPM based on the PSA FF-A specification (*Secure Partition Manager*).
- SPM based on the MM interface.

Both implementations differ in their architectures and only one can be selected at build time.

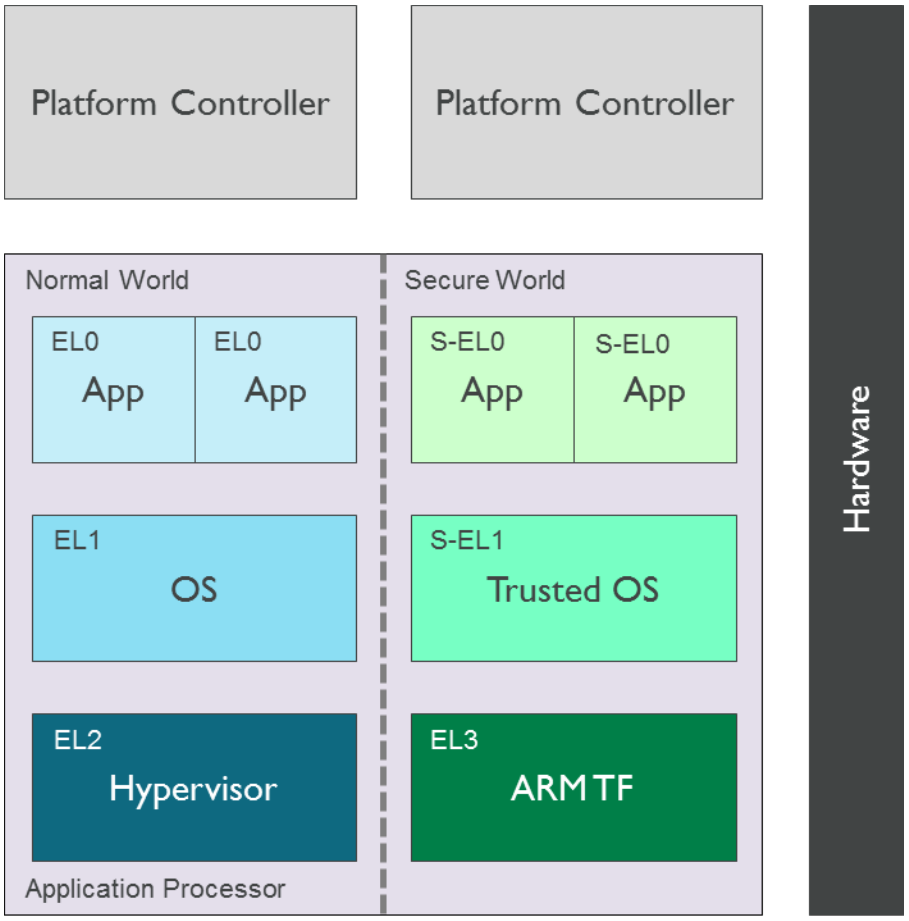
This document describes the latter implementation where the Secure Partition Manager resides at EL3 and management services run from isolated Secure Partitions at S-EL0. The communication protocol is established through the Management Mode (MM) interface.

4.13.2 Background

In some market segments that primarily deal with client-side devices like mobile phones, tablets, STBs and embedded devices, a Trusted OS instantiates trusted applications to provide security services like DRM, secure payment and authentication. The Global Platform TEE Client API specification defines the API used by Non-secure world applications to access these services. A Trusted OS fulfils the requirements of a security service as described above.

Management services are typically implemented at the highest level of privilege in the system, i.e. EL3 in Trusted Firmware-A (TF-A). The service requirements are fulfilled by the execution environment provided by TF-A.

The following diagram illustrates the corresponding software stack:



In other market segments that primarily deal with server-side devices (e.g. data centres and enterprise servers) the secure software stack typically does not include a Global Platform Trusted OS. Security functions are accessed through other interfaces (e.g. ACPI TCG TPM interface, UEFI runtime variable service).

Placement of management and security functions with diverse requirements in a privileged Exception Level (i.e. EL3 or S-EL1) makes security auditing of firmware more difficult and does not allow isolation of unrelated services from each other either.

4.13.3 Introduction

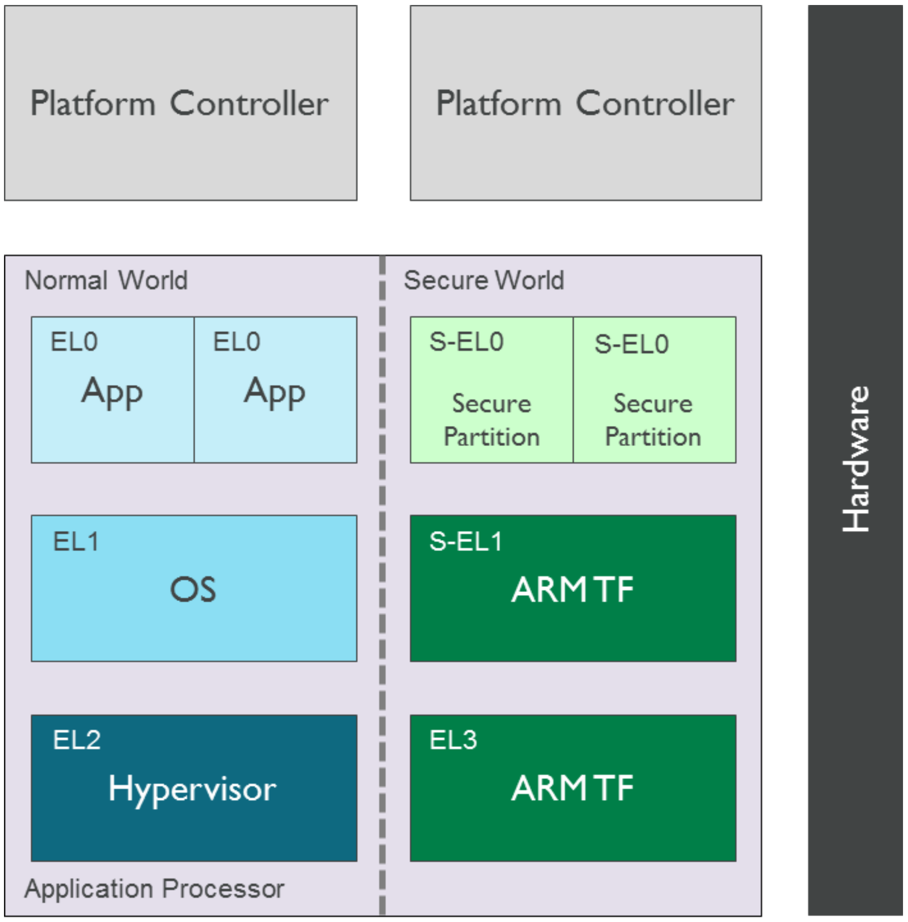
A **Secure Partition** is a software execution environment instantiated in S-EL0 that can be used to implement simple management and security services. Since S-EL0 is an unprivileged Exception Level, a Secure Partition relies on privileged firmware (i.e. TF-A) to be granted access to system and processor resources. Essentially, it is a software sandbox in the Secure world that runs under the control of privileged software, provides one or more services and accesses the following system resources:

- Memory and device regions in the system address map.
- PE system registers.
- A range of synchronous exceptions (e.g. SMC function identifiers).

Note that currently TF-A only supports handling one Secure Partition.

A Secure Partition enables TF-A to implement only the essential secure services in EL3 and instantiate the rest in a partition in S-EL0. Furthermore, multiple Secure Partitions can be used to isolate unrelated services from each other.

The following diagram illustrates the place of a Secure Partition in a typical Armv8-A software stack. A single or multiple Secure Partitions provide secure services to software components in the Non-secure world and other Secure Partitions.



The TF-A build system is responsible for including the Secure Partition image in the FIP. During boot, BL2 includes support to authenticate and load the Secure Partition image. A BL31 component called **Secure Partition Manager (SPM)** is responsible for managing the partition. This is semantically similar to a hypervisor managing a virtual machine.

The SPM is responsible for the following actions during boot:

- Allocate resources requested by the Secure Partition.
- Perform architectural and system setup required by the Secure Partition to fulfil a service request.
- Implement a standard interface that is used for initialising a Secure Partition.

The SPM is responsible for the following actions during runtime:

- Implement a standard interface that is used by a Secure Partition to fulfil service requests.
- Implement a standard interface that is used by the Non-secure world for accessing the services exported by a Secure Partition. A service can be invoked through a SMC.

Alternatively, a partition can be viewed as a thread of execution running under the control of the SPM. Hence common programming concepts described below are applicable to a partition.

4.13.4 Description

The previous section introduced some general aspects of the software architecture of a Secure Partition. This section describes the specific choices made in the current implementation of this software architecture. Subsequent revisions of the implementation will include a richer set of features that enable a more flexible architecture.

Building TF-A with Secure Partition support

SPM is supported on the Arm FVP exclusively at the moment. The current implementation supports inclusion of only a single Secure Partition in which a service always runs to completion (e.g. the requested services cannot be preempted to give control back to the Normal world).

It is not currently possible for BL31 to integrate SPM support and a Secure Payload Dispatcher (SPD) at the same time; they are mutually exclusive. In the SPM bootflow, a Secure Partition image executing at S-EL0 replaces the Secure Payload image executing at S-EL1 (e.g. a Trusted OS). Both are referred to as BL32.

A working prototype of a SP has been implemented by re-purposing the EDK2 code and tools, leveraging the concept of the *Standalone Management Mode (MM)* in the UEFI specification (see the PI v1.6 Volume 4: Management Mode Core Interface). This will be referred to as the *Standalone MM Secure Partition* in the rest of this document.

To enable SPM support in TF-A, the source code must be compiled with the build flag `SPM_MM=1`, along with `EL3_EXCEPTION_HANDLING=1`. On Arm platforms the build option `ARM_BL31_IN_DRAM` must be set to 1. Also, the location of the binary that contains the BL32 image (`BL32=path/to/image.bin`) must be specified.

First, build the Standalone MM Secure Partition. To build it, refer to the instructions in the EDK2 repository.

Then build TF-A with SPM support and include the Standalone MM Secure Partition image in the FIP:

```
BL32=path/to/standalone/mm/sp BL33=path/to/bl33.bin \
make PLAT=fvp SPM_MM=1 EL3_EXCEPTION_HANDLING=1 ARM_BL31_IN_DRAM=1 all fip
```

Describing Secure Partition resources

TF-A exports a porting interface that enables a platform to specify the system resources required by the Secure Partition. Some instructions are given below. However, this interface is under development and it may change as new features are implemented.

- A Secure Partition is considered a BL32 image, so the same defines that apply to BL32 images apply to a Secure Partition: `BL32_BASE` and `BL32_LIMIT`.
- The following defines are needed to allocate space for the translation tables used by the Secure Partition: `PLAT_SP_IMAGE_MMAP_REGIONS` and `PLAT_SP_IMAGE_MAX_XLAT_TABLES`.
- The functions `plat_get_secure_partition_mmap()` and `plat_get_secure_partition_boot_info()` have to be implemented. The file `plat/arm/board/fvp/fvp_common.c` can be used as an example. It uses the defines in `include/plat/arm/common/arm_spm_def.h`.
  - `plat_get_secure_partition_mmap()` returns an array of mmap regions that describe the memory regions that the SPM needs to allocate for a Secure Partition.
  - `plat_get_secure_partition_boot_info()` returns a `spm_mm_boot_info_t` struct that is populated by the platform with information about the memory map of the Secure Partition.

For an example of all the changes in context, you may refer to commit `e29efeb1b4`, in which the port for FVP was introduced.

Accessing Secure Partition services

The SMC Calling Convention (*Arm DEN 0028B*) describes SMCs as a conduit for accessing services implemented in the Secure world. The `MM_COMMUNICATE` interface defined in the Management Mode Interface Specification (*Arm DEN 0060A*) is used to invoke a Secure Partition service as a Fast Call.

The mechanism used to identify a service within the partition depends on the service implementation. It is assumed that the caller of the service will be able to discover this mechanism through standard platform discovery mechanisms like ACPI and Device Trees. For example, *Volume 4: Platform Initialisation Specification v1.6. Management Mode Core Interface* specifies that a GUID is used to identify a management mode service. A client populates the GUID in the `EFI_MM_COMMUNICATE_HEADER`. The header is populated in the communication buffer shared with the Secure Partition.

A Fast Call appears to be atomic from the perspective of the caller and returns when the requested operation has completed. A service invoked through the `MM_COMMUNICATE` SMC will run to completion in the partition on a given CPU. The SPM is responsible for guaranteeing this behaviour. This means that there can only be a single outstanding Fast Call in a partition on a given CPU.

Exchanging data with the Secure Partition

The exchange of data between the Non-secure world and the partition takes place through a shared memory region. The location of data in the shared memory area is passed as a parameter to the `MM_COMMUNICATE` SMC. The shared memory area is statically allocated by the SPM and is expected to be either implicitly known to the Non-secure world or discovered through a platform discovery mechanism e.g. ACPI table or device tree. It is possible for the Non-secure world to exchange data with a partition only if it has been populated in this shared memory area. The shared memory area is implemented as per the guidelines specified in Section 3.2.3 of the Management Mode Interface Specification (*Arm DEN 0060A*).

The format of data structures used to encapsulate data in the shared memory is agreed between the Non-secure world and the Secure Partition. For example, in the Management Mode Interface specification (*Arm DEN 0060A*), Section 4 describes that the communication buffer shared between the Non-secure world and the Management Mode (MM) in the Secure world must be of the type `EFI_MM_COMMUNICATE_HEADER`. This data structure is defined in *Volume 4: Platform Initialisation Specification v1.6. Management Mode Core Interface*. Any caller of a MM service will have to use the `EFI_MM_COMMUNICATE_HEADER` data structure.

4.13.5 Runtime model of the Secure Partition

This section describes how the Secure Partition interfaces with the SPM.

Interface with SPM

In order to instantiate one or more secure services in the Secure Partition in S-EL0, the SPM should define the following types of interfaces:

- Interfaces that enable access to privileged operations from S-EL0. These operations typically require access to system resources that are either shared amongst multiple software components in the Secure world or cannot be directly accessed from an unprivileged Exception Level.
- Interfaces that establish the control path between the SPM and the Secure Partition.

This section describes the APIs currently exported by the SPM that enable a Secure Partition to initialise itself and export its services in S-EL0. These interfaces are not accessible from the Non-secure world.

Conduit

The SMC Calling Convention (*Arm DEN 0028B*) specification describes the SMC and HVC conduits for accessing firmware services and their availability depending on the implemented Exception levels. In S-EL0, the Supervisor Call exception (SVC) is the only architectural mechanism available for unprivileged software to make a request for an operation implemented in privileged software. Hence, the SVC conduit must be used by the Secure Partition to access interfaces implemented by the SPM.

A SVC causes an exception to be taken to S-EL1. TF-A assumes ownership of S-EL1 and installs a simple exception vector table in S-EL1 that relays a SVC request from a Secure Partition as a SMC request to the SPM in EL3. Upon servicing the SMC request, Trusted Firmware-A returns control directly to S-EL0 through an ERET instruction.

Calling conventions

The SMC Calling Convention (*Arm DEN 0028B*) specification describes the 32-bit and 64-bit calling conventions for the SMC and HVC conduits. The SVC conduit introduces the concept of SVC32 and SVC64 calling conventions. The SVC32 and SVC64 calling conventions are equivalent to the 32-bit (SMC32) and the 64-bit (SMC64) calling conventions respectively.

Communication initiated by SPM

A service request is initiated from the SPM through an exception return instruction (ERET) to S-EL0. Later, the Secure Partition issues an SVC instruction to signal completion of the request. Some example use cases are given below:

- A request to initialise the Secure Partition during system boot.
- A request to handle a runtime service request.

Communication initiated by Secure Partition

A request is initiated from the Secure Partition by executing a SVC instruction. An ERET instruction is used by TF-A to return to S-EL0 with the result of the request.

For instance, a request to perform privileged operations on behalf of a partition (e.g. management of memory attributes in the translation tables for the Secure EL1&0 translation regime).

Interfaces

The current implementation reserves function IDs for Fast Calls in the Standard Secure Service calls range (see SMC Calling Convention (*Arm DEN 0028B*) specification) for each API exported by the SPM. This section defines the function prototypes for each function ID. The function IDs specify whether one or both of the SVC32 and SVC64 calling conventions can be used to invoke the corresponding interface.

Secure Partition Event Management

The Secure Partition provides an Event Management interface that is used by the SPM to delegate service requests to the Secure Partition. The interface also allows the Secure Partition to:

- Register with the SPM a service that it provides.
- Indicate completion of a service request delegated by the SPM

Miscellaneous interfaces

`SPM_MM_VERSION_AARCH32`

- Description

  Returns the version of the interface exported by SPM.

- Parameters

  – **uint32** - Function ID

    * SVC32 Version: **0x84000060**

- Return parameters

  – **int32** - Status

    On success, the format of the value is as follows:

    * Bit [31]: Must be 0
    * Bits [30:16]: Major Version. Must be 0 for this revision of the SPM interface.
    * Bits [15:0]: Minor Version. Must be 1 for this revision of the SPM interface.

    On error, the format of the value is as follows:

    * `NOT_SUPPORTED`: SPM interface is not supported or not available for the client.

- Usage

  This function returns the version of the Secure Partition Manager implementation. The major version is 0 and the minor version is 1. The version number is a 31-bit unsigned integer, with the upper 15 bits denoting the major revision, and the lower 16 bits denoting the minor revision. The following rules apply to the version numbering:

  – Different major revision values indicate possibly incompatible functions.

  – For two revisions, A and B, for which the major revision values are identical, if the minor revision value of revision B is greater than the minor revision value of revision A, then every function in revision A must work in a compatible way with revision B. However, it is possible for revision B to have a higher function count than revision A.

- Implementation responsibilities

  If this function returns a valid version number, all the functions that are described subsequently must be implemented, unless it is explicitly stated that a function is optional.

See *Error Codes* for integer values that are associated with each return code.

Secure Partition Initialisation

The SPM is responsible for initialising the architectural execution context to enable initialisation of a service in S-EL0. The responsibilities of the SPM are listed below. At the end of initialisation, the partition issues a `MM_SP_EVENT_COMPLETE_AARCH64` call (described later) to signal readiness for handling requests for services implemented by the Secure Partition. The initialisation event is executed as a Fast Call.

Entry point invocation

The entry point for service requests that should be handled as Fast Calls is used as the target of the ERET instruction to start initialisation of the Secure Partition.

Architectural Setup

At cold boot, system registers accessible from S-EL0 will be in their reset state unless otherwise specified. The SPM will perform the following architectural setup to enable execution in S-EL0

MMU setup

The platform port of a Secure Partition specifies to the SPM a list of regions that it needs access to and their attributes. The SPM validates this resource description and initialises the Secure EL1&0 translation regime as follows.

1. Device regions are mapped with nGnRE attributes and Execute Never instruction access permissions.
2. Code memory regions are mapped with RO data and Executable instruction access permissions.
3. Read Only data memory regions are mapped with RO data and Execute Never instruction access permissions.
4. Read Write data memory regions are mapped with RW data and Execute Never instruction access permissions.
5. If the resource description does not explicitly describe the type of memory regions then all memory regions will be marked with Code memory region attributes.
6. The `UXN` and `PXN` bits are set for regions that are not executable by S-EL0 or S-EL1.

System Register Setup

System registers that influence software execution in S-EL0 are setup by the SPM as follows:

1. `SCTLR_EL1`
    - `UCI=1`
    - `EOE=0`
    - `WXN=1`
    - `nTWE=1`
    - `nTWI=1`
    - `UCT=1`
    - `DZE=1`
    - `I=1`
    - `UMA=0`

   - `SA0=1`
   - `C=1`
   - `A=1`
   - `M=1`
2. `CPACR_EL1`
   - `FPEN=b'11`
3. `PSTATE`
   - `D,A,I,F=1`
   - `CurrentEL=0` (EL0)
   - `SpSel=0` (Thread mode)
   - `NRW=0` (AArch64)

General Purpose Register Setup

SPM will invoke the entry point of a service by executing an ERET instruction. This transition into S-EL0 is special since it is not in response to a previous request through a SVC instruction. This is the first entry into S-EL0. The general purpose register usage at the time of entry will be as specified in the "Return State" column of Table 3-1 in Section 3.1 "Register use in AArch64 SMC calls" of the SMC Calling Convention (*Arm DEN 0028B*) specification. In addition, certain other restrictions will be applied as described below.

1. `SP_EL0`

   A non-zero value will indicate that the SPM has initialised the stack pointer for the current CPU.

   The value will be 0 otherwise.
2. `X4-X30`

   The values of these registers will be 0.
3. `X0-X3`

   Parameters passed by the SPM.
   - `X0`: Virtual address of a buffer shared between EL3 and S-EL0. The buffer will be mapped in the Secure EL1&0 translation regime with read-only memory attributes described earlier.
   - `X1`: Size of the buffer in bytes.
   - `X2`: Cookie value (*IMPLEMENTATION DEFINED*).
   - `X3`: Cookie value (*IMPLEMENTATION DEFINED*).

Runtime Event Delegation

The SPM receives requests for Secure Partition services through a synchronous invocation (i.e. a SMC from the Non-secure world). These requests are delegated to the partition by programming a return from the last `MM_SP_EVENT_COMPLETE_AARCH64` call received from the partition. The last call was made to signal either completion of Secure Partition initialisation or completion of a partition service request.

`MM_SP_EVENT_COMPLETE_AARCH64`

- Description

  Signal completion of the last SP service request.

- Parameters

  - **uint32** - Function ID

    * SVC64 Version: **0xC4000061**

  - **int32** - Event Status Code

    Zero or a positive value indicates that the event was handled successfully. The values depend upon the original event that was delegated to the Secure partition. They are described as follows.

    * `SUCCESS` : Used to indicate that the Secure Partition was initialised or a runtime request was handled successfully.
    * Any other value greater than 0 is used to pass a specific Event Status code in response to a runtime event.

    A negative value indicates an error. The values of Event Status code depend on the original event.

- Return parameters

  - **int32** - Event ID/Return Code

    Zero or a positive value specifies the unique ID of the event being delegated to the partition by the SPM.

    In the current implementation, this parameter contains the function ID of the `MM_COMMUNICATE` SMC. This value indicates to the partition that an event has been delegated to it in response to an `MM_COMMUNICATE` request from the Non-secure world.

    A negative value indicates an error. The format of the value is as follows:

    * `NOT_SUPPORTED`: Function was called from the Non-secure world.

    See *Error Codes* for integer values that are associated with each return code.

  - **uint32** - Event Context Address

    Address of a buffer shared between the SPM and Secure Partition to pass event specific information. The format of the data populated in the buffer is implementation defined.

    The buffer is mapped in the Secure EL1&0 translation regime with read-only memory attributes described earlier.

    For the SVC64 version, this parameter is a 64-bit Virtual Address (VA).

    For the SVC32 version, this parameter is a 32-bit Virtual Address (VA).

  - **uint32** - Event context size

    Size of the memory starting at Event Address.

  - **uint32/uint64** - Event Cookie

    This is an optional parameter. If unused its value is SBZ.

- Usage

  This function signals to the SPM that the handling of the last event delegated to a partition has completed. The partition is ready to handle its next event. A return from this function is in response to the next event that will be delegated to the partition. The return parameters describe the next event.

- Caller responsibilities

  A Secure Partition must only call `MM_SP_EVENT_COMPLETE_AARCH64` to signal completion of a request that was delegated to it by the SPM.

- Callee responsibilities

  When the SPM receives this call from a Secure Partition, the corresponding syndrome information can be used to return control through an ERET instruction, to the instruction immediately after the call in the Secure Partition context. This syndrome information comprises of general purpose and system register values when the call was made.

  The SPM must save this syndrome information and use it to delegate the next event to the Secure Partition. The return parameters of this interface must specify the properties of the event and be populated in `X0-X3/W0-W3` registers.

Secure Partition Memory Management

A Secure Partition executes at S-EL0, which is an unprivileged Exception Level. The SPM is responsible for enabling access to regions of memory in the system address map from a Secure Partition. This is done by mapping these regions in the Secure EL1&0 Translation regime with appropriate memory attributes. Attributes refer to memory type, permission, cacheability and shareability attributes used in the Translation tables. The definitions of these attributes and their usage can be found in the Armv8-A ARM (*Arm DDI 0487*).

All memory required by the Secure Partition is allocated upfront in the SPM, even before handing over to the Secure Partition for the first time. The initial access permissions of the memory regions are statically provided by the platform port and should allow the Secure Partition to run its initialisation code.

However, they might not suit the final needs of the Secure Partition because its final memory layout might not be known until the Secure Partition initialises itself. As the Secure Partition initialises its runtime environment it might, for example, load dynamically some modules. For instance, a Secure Partition could implement a loader for a standard executable file format (e.g. an PE-COFF loader for loading executable files at runtime). These executable files will be a part of the Secure Partition image. The location of various sections in an executable file and their permission attributes (e.g. read-write data, read-only data and code) will be known only when the file is loaded into memory.

In this case, the Secure Partition needs a way to change the access permissions of its memory regions. The SPM provides this feature through the `MM_SP_MEMORY_ATTRIBUTES_SET_AARCH64` SVC interface. This interface is available to the Secure Partition during a specific time window: from the first entry into the Secure Partition up to the first `SP_EVENT_COMPLETE` call that signals the Secure Partition has finished its initialisation. Once the initialisation is complete, the SPM does not allow changes to the memory attributes.

This section describes the standard SVC interface that is implemented by the SPM to determine and change permission attributes of memory regions that belong to a Secure Partition.

MM_SP_MEMORY_ATTRIBUTES_GET_AARCH64

- Description

  Request the permission attributes of a memory region from S-EL0.

- Parameters

  - **uint32** Function ID

    * SVC64 Version: **0xC4000064**

  - **uint64** Base Address

    This parameter is a 64-bit Virtual Address (VA).

    There are no alignment restrictions on the Base Address. The permission attributes of the translation granule it lies in are returned.

- Return parameters

  - **int32** - Memory Attributes/Return Code

    On success the format of the Return Code is as follows:

    * Bits[1:0] : Data access permission

      · b'00 : No access

      · b'01 : Read-Write access

      · b'10 : Reserved

      · b'11 : Read-only access

    * Bit[2]: Instruction access permission

      · b'0 : Executable

      · b'1 : Non-executable

    * Bit[30:3] : Reserved. SBZ.

    * Bit[31] : Must be 0

    On failure the following error codes are returned:

    * `INVALID_PARAMETERS`: The Secure Partition is not allowed to access the memory region the Base Address lies in.

    * `NOT_SUPPORTED` : The SPM does not support retrieval of attributes of any memory page that is accessible by the Secure Partition, or the function was called from the Non-secure world. Also returned if it is used after `MM_SP_EVENT_COMPLETE_AARCH64`.

    See *Error Codes* for integer values that are associated with each return code.

- Usage

  This function is used to request the permission attributes for S-EL0 on a memory region accessible from a Secure Partition. The size of the memory region is equal to the Translation Granule size used in the Secure EL1&0 translation regime. Requests to retrieve other memory region attributes are not currently supported.

- Caller responsibilities

  The caller must obtain the Translation Granule Size of the Secure EL1&0 translation regime from the SPM through an implementation defined method.

- Callee responsibilities

  The SPM must not return the memory access controls for a page of memory that is not accessible from a Secure Partition.

`MM_SP_MEMORY_ATTRIBUTES_SET_AARCH64`

- Description

  Set the permission attributes of a memory region from S-EL0.

- Parameters
  - **uint32** - Function ID
    - SVC64 Version: **0xC4000065**
  - **uint64** - Base Address

    This parameter is a 64-bit Virtual Address (VA).

    The alignment of the Base Address must be greater than or equal to the size of the Translation Granule Size used in the Secure EL1&0 translation regime.
  - **uint32** - Page count

    Number of pages starting from the Base Address whose memory attributes should be changed. The page size is equal to the Translation Granule Size.
  - **uint32** - Memory Access Controls
    - Bits[1:0] : Data access permission
      - b'00 : No access
      - b'01 : Read-Write access
      - b'10 : Reserved
      - b'11 : Read-only access
    - Bit[2] : Instruction access permission
      - b'0 : Executable
      - b'1 : Non-executable
    - Bits[31:3] : Reserved. SBZ.

    A combination of attributes that mark the region with RW and Executable permissions is prohibited. A request to mark a device memory region with Executable permissions is prohibited.

- Return parameters
  - **int32** - Return Code
    - `SUCCESS`: The Memory Access Controls were changed successfully.
    - `DENIED`: The SPM is servicing a request to change the attributes of a memory region that overlaps with the region specified in this request.
    - `INVALID_PARAMETER`:An invalid combination of Memory Access Controls has been specified. The Base Address is not correctly aligned. The Secure Partition is not allowed to access part or all of the memory region specified in the call.

    * `NO_MEMORY`: The SPM does not have memory resources to change the attributes of the memory region in the translation tables.
    * `NOT_SUPPORTED`: The SPM does not permit change of attributes of any memory region that is accessible by the Secure Partition. Function was called from the Non-secure world. Also returned if it is used after `MM_SP_EVENT_COMPLETE_AARCH64`.

  See *Error Codes* for integer values that are associated with each return code.

- Usage

  This function is used to change the permission attributes for S-EL0 on a memory region accessible from a Secure Partition. The size of the memory region is equal to the Translation Granule size used in the Secure EL1&0 translation regime. Requests to change other memory region attributes are not currently supported.

  This function is only available at boot time. This interface is revoked after the Secure Partition sends the first `MM_SP_EVENT_COMPLETE_AARCH64` to signal that it is initialised and ready to receive run-time requests.

- Caller responsibilities

  The caller must obtain the Translation Granule Size of the Secure EL1&0 translation regime from the SPM through an implementation defined method.

- Callee responsibilities

  The SPM must preserve the original memory access controls of the region of memory in case of an unsuccessful call. The SPM must preserve the consistency of the S-EL1 translation regime if this function is called on different PEs concurrently and the memory regions specified overlap.

Error Codes

Name	Value
`SUCCESS`	0
`NOT_SUPPORTED`	-1
`INVALID_PARAMETER`	-2
`DENIED`	-3
`NO_MEMORY`	-5
`NOT_PRESENT`	-7

4.14 PSA FF-A manifest binding to device tree

This document defines the nodes and properties used to define a partition, according to the PSA FF-A specification.

4.14.1 Version 1.0

Partition Properties

- **compatible [mandatory]**
  - value type: <string>
  - Must be the string "arm,ffa-manifest-X.Y" which specifies the major and minor versions fo the device tree binding for the FFA manifest represented by this node. The minor number is incremented if the binding changes in a backwards compatible manner.
    - X is an integer representing the major version number of this document.
    - Y is an integer representing the minor version number of this document.
- **ffa-version [mandatory]**
  - value type: <u32>
  - Must be two 16 bits values (X, Y), concatenated as 31:16 -> X, 15:0 -> Y, where:
    - X is the major version of PSA-FF-A expected by the partition at the FFA instance it will execute.
    - Y is the minor version of PSA-FF-A expected by the partition at the FFA instance it will execute.
- **uuid [mandatory]**
  - value type: <prop-encoded-array>
  - An array consisting of 4 <u32> values, identifying the UUID of the service implemented by this partition. The UUID format is described in RFC 4122.
- **id**
  - value type: <u32>
  - Pre-allocated partition ID.
- **auxiliary-id**
  - value type: <u32>
  - Pre-allocated ID that could be used in memory management transactions.
- **description**
  - value type: <string>
  - Name of the partition e.g. for debugging purposes.
- **execution-ctx-count [mandatory]**
  - value type: <u32>
  - Number of vCPUs that a VM or SP wants to instantiate.
    - In the absence of virtualization, this is the number of execution contexts that a partition implements.
    - If value of this field = 1 and number of PEs > 1 then the partition is treated as UP & migrate capable.
    - If the value of this field > 1 then the partition is treated as a MP capable partition irrespective of the number of PEs.
- **exception-level [mandatory]**

  - value type: <u32>
  - The target exception level for the partition:
    * 0x0: EL1
    * 0x1: S_EL0
    * 0x2: S_EL1
- **execution-state [mandatory]**
  - value type: <u32>
  - The target execution state of the partition:
    * 0: AArch64
    * 1: AArch32
- **load-address**
  - value type: <u64>
  - Physical base address of the partition in memory. Absence of this field indicates that the partition is position independent and can be loaded at any address chosen at boot time.
- **entrypoint-offset**
  - value type: <u64>
  - Offset from the base of the partition's binary image to the entry point of the partition. Absence of this field indicates that the entry point is at offset 0x0 from the base of the partition's binary.
- **xlat-granule [mandatory]**
  - value type: <u32>
  - Translation granule used with the partition:
    * 0x0: 4k
    * 0x1: 16k
    * 0x2: 64k
- **boot-order**
  - value type: <u32>
  - A unique number amongst all partitions that specifies if this partition must be booted before others. The partition with the smaller number will be booted first.
- **rx-tx-buffer**
  - value type: "memory-regions" node
  - Specific "memory-regions" nodes that describe the RX/TX buffers expected by the partition. The "compatible" must be the string "arm,ffa-manifest-rx_tx-buffer".
- **messaging-method [mandatory]**
  - value type: <u32>
  - Specifies which messaging methods are supported by the partition:
    * 0x0: direct messaging method
    * 0x1: indirect messaging method

    * 0x2: both direct and indirect messaging methods

- **has-primary-scheduler**
    - value type: <empty>
    - Presence of this field indicates that the partition implements the primary scheduler. If so, run-time EL must be EL1.
- **run-time-model**
    - value type: <u32>
    - Run time model that the SPM must enforce for this SP:
        * 0x0: Run to completion
        * 0x1: Preemptible
- **time-slice-mem**
    - value type: <empty>
    - Presence of this field indicates that the partition doesn't expect the partition manager to time slice long running memory management functions.
- **gp-register-num**
    - value type: <u32>
    - Presence of this field indicates that the partition expects the ffa_init_info structure to be passed in via the specified general purpose register. The field specifies the general purpose register number but not its width. The width is derived from the partition's execution state, as specified in the partition properties. For example, if the number value is 1 then the general-purpose register used will be x1 in AArch64 state and w1 in AArch32 state.
- **stream-endpoint-ids**
    - value type: <prop-encoded-array>
    - List of <u32> tuples, identifying the IDs this partition is acting as proxy for.

4.14.2 Memory Regions

- **compatible [mandatory]**
    - value type: <string>
    - Must be the string "arm,ffa-manifest-memory-regions".
- **description**
    - value type: <string>
    - Name of the memory region e.g. for debugging purposes.
- **pages-count [mandatory]**
    - value type: <u32>
    - Count of pages of memory region as a multiple of the translation granule size
- **attributes [mandatory]**
    - value type: <u32>
    - Mapping modes: ORed to get required permission

    * 0x1: Read
    * 0x2: Write
    * 0x4: Execute

- **base-address**
  - value type: <u64>
  - Base address of the region. The address must be aligned to the translation granule size. The address given may be a Physical Address (PA), Virtual Address (VA), or Intermediate Physical Address (IPA). Refer to the FFA specification for more information on the restrictions around the address type. If the base address is omitted then the partition manager must map a memory region of the specified size into the partition's translation regime and then communicate the region properties (including the base address chosen by the partition manager) to the partition.

4.14.3 Device Regions

- **compatible [mandatory]**
  - value type: <string>
  - Must be the string "arm,ffa-manifest-device-regions".
- **description**
  - value type: <string>
  - Name of the device region e.g. for debugging purposes.
- **reg [mandatory]**
  - value type: <prop-encoded-array>
  - A (address, num-pages) pair describing the device, where:
    * address: The physical base address <u64> value of the device MMIO region.
    * num-pages: The <u32> number of pages of the region. The total size of the region is this value multiplied by the translation granule size.
- **attributes [mandatory]**
  - value type: <u32>
  - Mapping modes: ORed to get required permission
    * 0x1: Read
    * 0x2: Write
    * 0x4: Execute
- **smmu-id**
  - value type: <u32>
  - On systems with multiple System Memory Management Units (SMMUs) this identifier is used to inform the partition manager which SMMU the device is upstream of. If the field is omitted then it is assumed that the device is not upstream of any SMMU.
- **stream-ids**
  - value type: <prop-encoded-array>
  - A list of (id, mem-manage) pair, where:

    * id: A unique <u32> value amongst all devices assigned to the partition.

- **interrupts [mandatory]**
  - value type: <prop-encoded-array>
  - A list of (id, attributes) pair describing the device interrupts, where:
    * id: The <u32> interrupt IDs.
    * attributes: A <u32> value, containing the attributes for each interrupt ID:
      · Interrupt type: SPI, PPI, SGI
      · Interrupt configuration: Edge triggered, Level triggered
      · Interrupt security state: Secure, Non-secure
      · Interrupt priority value
      · Target execution context/vCPU for each SPI
- **exclusive-access**
  - value type: <empty>
  - Presence of this field implies that this endpoint must be granted exclusive access and ownership of this devices's MMIO region.

4.15 Translation (XLAT) Tables Library

This document describes the design of the translation tables library (version 2) used by Trusted Firmware-A (TF-A). This library provides APIs to create page tables based on a description of the memory layout, as well as setting up system registers related to the Memory Management Unit (MMU) and performing the required Translation Lookaside Buffer (TLB) maintenance operations.

More specifically, some use cases that this library aims to support are:

1. Statically allocate translation tables and populate them (at run-time) based on a description of the memory layout. The memory layout is typically provided by the platform port as a list of memory regions;
2. Support for generating translation tables pertaining to a different translation regime than the exception level the library code is executing at;
3. Support for dynamic mapping and unmapping of regions, even while the MMU is on. This can be used to temporarily map some memory regions and unmap them later on when no longer needed;
4. Support for non-identity virtual to physical mappings to compress the virtual address space;
5. Support for changing memory attributes of memory regions at run-time.

4.15.1 About version 1 and version 2

This document focuses on version 2 of the library, whose sources are available in the `lib/xlat_tables_v2` directory. Version 1 of the library can still be found in `lib/xlat_tables` directory but it is less flexible and doesn't support dynamic mapping. Although potential bug fixes will be applied to both versions, future features enhancements will focus on version 2 and might not be back-ported to version 1. Therefore, it is recommended to use version 2, especially for new platform ports.

However, please note that version 2 is still in active development and is not considered stable yet. Hence, compatibility breaks might be introduced.

From this point onwards, this document will implicitly refer to version 2 of the library.

4.15.2 Design concepts and interfaces

This section presents some of the key concepts and data structures used in the translation tables library.

mmap regions

An `mmap_region` is an abstract, concise way to represent a memory region to map. It is one of the key interfaces to the library. It is identified by:

- its physical base address;
- its virtual base address;
- its size;
- its attributes;
- its mapping granularity (optional).

See the `struct mmap_region` type in `xlat_tables_v2.h`.

The user usually provides a list of such mmap regions to map and lets the library transpose that in a set of translation tables. As a result, the library might create new translation tables, update or split existing ones.

The region attributes specify the type of memory (for example device or cached normal memory) as well as the memory access permissions (read-only or read-write, executable or not, secure or non-secure, and so on). In the case of the EL1&0 translation regime, the attributes also specify whether the region is a User region (EL0) or Privileged region (EL1). See the `MT_xxx` definitions in `xlat_tables_v2.h`. Note that for the EL1&0 translation regime the Execute Never attribute is set simultaneously for both EL1 and EL0.

The granularity controls the translation table level to go down to when mapping the region. For example, assuming the MMU has been configured to use a 4KB granule size, the library might map a 2MB memory region using either of the two following options:

- using a single level-2 translation table entry;
- using a level-2 intermediate entry to a level-3 translation table (which contains 512 entries, each mapping 4KB).

The first solution potentially requires less translation tables, hence potentially less memory. However, if part of this 2MB region is later remapped with different memory attributes, the library might need to split the existing page tables to refine the mappings. If a single level-2 entry has been used here, a level-3 table will need to be allocated on the fly and the level-2 modified to point to this new level-3 table. This has a performance cost at run-time.

If the user knows upfront that such a remapping operation is likely to happen then they might enforce a 4KB mapping granularity for this 2MB region from the beginning; remapping some of these 4KB pages on the fly then becomes a lightweight operation.

The region's granularity is an optional field; if it is not specified the library will choose the mapping granularity for this region as it sees fit (more details can be found in *The memory mapping algorithm* section below).

Translation Context

The library can create or modify translation tables pertaining to a different translation regime than the exception level the library code is executing at. For example, the library might be used by EL3 software (for instance BL31) to create translation tables pertaining to the S-EL1&0 translation regime.

This flexibility comes from the use of *translation contexts*. A *translation context* constitutes the superset of information used by the library to track the status of a set of translation tables for a given translation regime.

The library internally allocates a default translation context, which pertains to the translation regime of the current exception level. Additional contexts may be explicitly allocated and initialized using the `REGISTER_XLAT_CONTEXT()` macro. Separate APIs are provided to act either on the default translation context or on an alternative one.

To register a translation context, the user must provide the library with the following information:

- A name.

  The resulting translation context variable will be called after this name, to which `_xlat_ctx` is appended. For example, if the macro name parameter is `foo`, the context variable name will be `foo_xlat_ctx`.

- The maximum number of *mmap* regions to map.

  Should account for both static and dynamic regions, if applicable.

- The number of sub-translation tables to allocate.

  Number of translation tables to statically allocate for this context, excluding the initial lookup level translation table, which is always allocated. For example, if the initial lookup level is 1, this parameter would specify the number of level-2 and level-3 translation tables to pre-allocate for this context.

- The size of the virtual address space.

  Size in bytes of the virtual address space to map using this context. This will incidentally determine the number of entries in the initial lookup level translation table : the library will allocate as many entries as is required to map the entire virtual address space.

- The size of the physical address space.

  Size in bytes of the physical address space to map using this context.

The default translation context is internally initialized using information coming (for the most part) from platform-specific defines:

- name: hard-coded to `tf` ; hence the name of the default context variable is `tf_xlat_ctx`;
- number of *mmap* regions: `MAX_MMAP_REGIONS`;
- number of sub-translation tables: `MAX_XLAT_TABLES`;
- size of the virtual address space: `PLAT_VIRT_ADDR_SPACE_SIZE`;
- size of the physical address space: `PLAT_PHY_ADDR_SPACE_SIZE`.

Please refer to the *Porting Guide* for more details about these macros.

Static and dynamic memory regions

The library optionally supports dynamic memory mapping. This feature may be enabled using the `PLAT_XLAT_TABLES_DYNAMIC` platform build flag.

When dynamic memory mapping is enabled, the library categorises mmap regions as *static* or *dynamic*.

- *Static regions* are fixed for the lifetime of the system. They can only be added early on, before the translation tables are created and populated. They cannot be removed afterwards.
- *Dynamic regions* can be added or removed any time.

When the dynamic memory mapping feature is disabled, only static regions exist.

The dynamic memory mapping feature may be used to map and unmap transient memory areas. This is useful when the user needs to access some memory for a fixed period of time, after which the memory may be discarded and reclaimed. For example, a memory region that is only required at boot time while the system is initializing, or to temporarily share a memory buffer between the normal world and trusted world. Note that it is up to the caller to ensure that these regions are not accessed concurrently while the regions are being added or removed.

Although this feature provides some level of dynamic memory allocation, this does not allow dynamically allocating an arbitrary amount of memory at an arbitrary memory location. The user is still required to declare at compile-time the limits of these allocations ; the library will deny any mapping request that does not fit within this pre-allocated pool of memory.

4.15.3 Library APIs

The external APIs exposed by this library are declared and documented in the `xlat_tables_v2.h` header file. This should be the reference point for getting information about the usage of the different APIs this library provides. This section just provides some extra details and clarifications.

Although the `mmap_region` structure is a publicly visible type, it is not recommended to populate these structures by hand. Instead, wherever APIs expect function arguments of type `mmap_region_t`, these should be constructed using the `MAP_REGION*()` family of helper macros. This is to limit the risk of compatibility breaks, should the `mmap_region` structure type evolve in the future.

The `MAP_REGION()` and `MAP_REGION_FLAT()` macros do not allow specifying a mapping granularity, which leaves the library implementation free to choose it. However, in cases where a specific granularity is required, the `MAP_REGION2()` macro might be used instead.

As explained earlier in this document, when the dynamic mapping feature is disabled, there is no notion of dynamic regions. Conceptually, there are only static regions. For this reason (and to retain backward compatibility with the version 1 of the library), the APIs that map static regions do not embed the word *static* in their functions names (for example `mmap_add_region()`), in contrast with the dynamic regions APIs (for example `mmap_add_dynamic_region()`).

Although the definition of static and dynamic regions is not based on the state of the MMU, the two are still related in some way. Static regions can only be added before `init_xlat_tables()` is called and `init_xlat_tables()` must be called while the MMU is still off. As a result, static regions cannot be added once the MMU has been enabled. Dynamic regions can be added with the MMU on or off. In practice, the usual call flow would look like this:

1. The MMU is initially off.
2. Add some static regions, add some dynamic regions.
3. Initialize translation tables based on the list of mmap regions (using one of the `init_xlat_tables*()` APIs).
4. At this point, it is no longer possible to add static regions. Dynamic regions can still be added or removed.

5. Enable the MMU.
6. Dynamic regions can continue to be added or removed.

Because static regions are added early on at boot time and are all in the control of the platform initialization code, the `mmap_add*()` family of APIs are not expected to fail. They do not return any error code.

Nonetheless, these APIs will check upfront whether the region can be successfully added before updating the translation context structure. If the library detects that there is insufficient memory to meet the request, or that the new region will overlap another one in an invalid way, or if any other unexpected error is encountered, they will print an error message on the UART. Additionally, when asserts are enabled (typically in debug builds), an assertion will be triggered. Otherwise, the function call will just return straight away, without adding the offending memory region.

4.15.4 Library limitations

Dynamic regions are not allowed to overlap each other. Static regions are allowed to overlap as long as one of them is fully contained inside the other one. This is allowed for backwards compatibility with the previous behaviour in the version 1 of the library.

4.15.5 Implementation details

Code structure

The library is divided into 4 modules:

- **Core module**

  Provides the main functionality of the library, such as the initialization of translation tables contexts and mapping/unmapping memory regions. This module provides functions such as `mmap_add_region_ctx` that let the caller specify the translation tables context affected by them.

  See `xlat_tables_core.c`.

- **Active context module**

  Instantiates the context that is used by the current BL image and provides helpers to manipulate it, abstracting it from the rest of the code. This module provides functions such as `mmap_add_region`, that directly affect the BL image using them.

  See `xlat_tables_context.c`.

- **Utilities module**

  Provides additional functionality like debug print of the current state of the translation tables and helpers to query memory attributes and to modify them.

  See `xlat_tables_utils.c`.

- **Architectural module**

  Provides functions that are dependent on the current execution state (AArch32/AArch64), such as the functions used for TLB invalidation, setup the MMU, or calculate the Physical Address Space size. They do not need a translation context to work on.

  See `aarch32/xlat_tables_arch.c` and `aarch64/xlat_tables_arch.c`.

From mmap regions to translation tables

A translation context contains a list of `mmap_region_t`, which holds the information of all the regions that are mapped at any given time. Whenever there is a request to map (resp. unmap) a memory region, it is added to (resp. removed from) the `mmap_region_t` list.

The mmap regions list is a conceptual way to represent the memory layout. At some point, the library has to convert this information into actual translation tables to program into the MMU.

Before the `init_xlat_tables()` API is called, the library only acts on the mmap regions list. Adding a static or dynamic region at this point through one of the `mmap_add*()` APIs does not affect the translation tables in any way, they only get registered in the internal mmap region list. It is only when the user calls the `init_xlat_tables()` that the translation tables are populated in memory based on the list of mmap regions registered so far. This is an optimization that allows creation of the initial set of translation tables in one go, rather than having to edit them every time while the MMU is disabled.

After the `init_xlat_tables()` API has been called, only dynamic regions can be added. Changes to the translation tables (as well as the mmap regions list) will take effect immediately.

The memory mapping algorithm

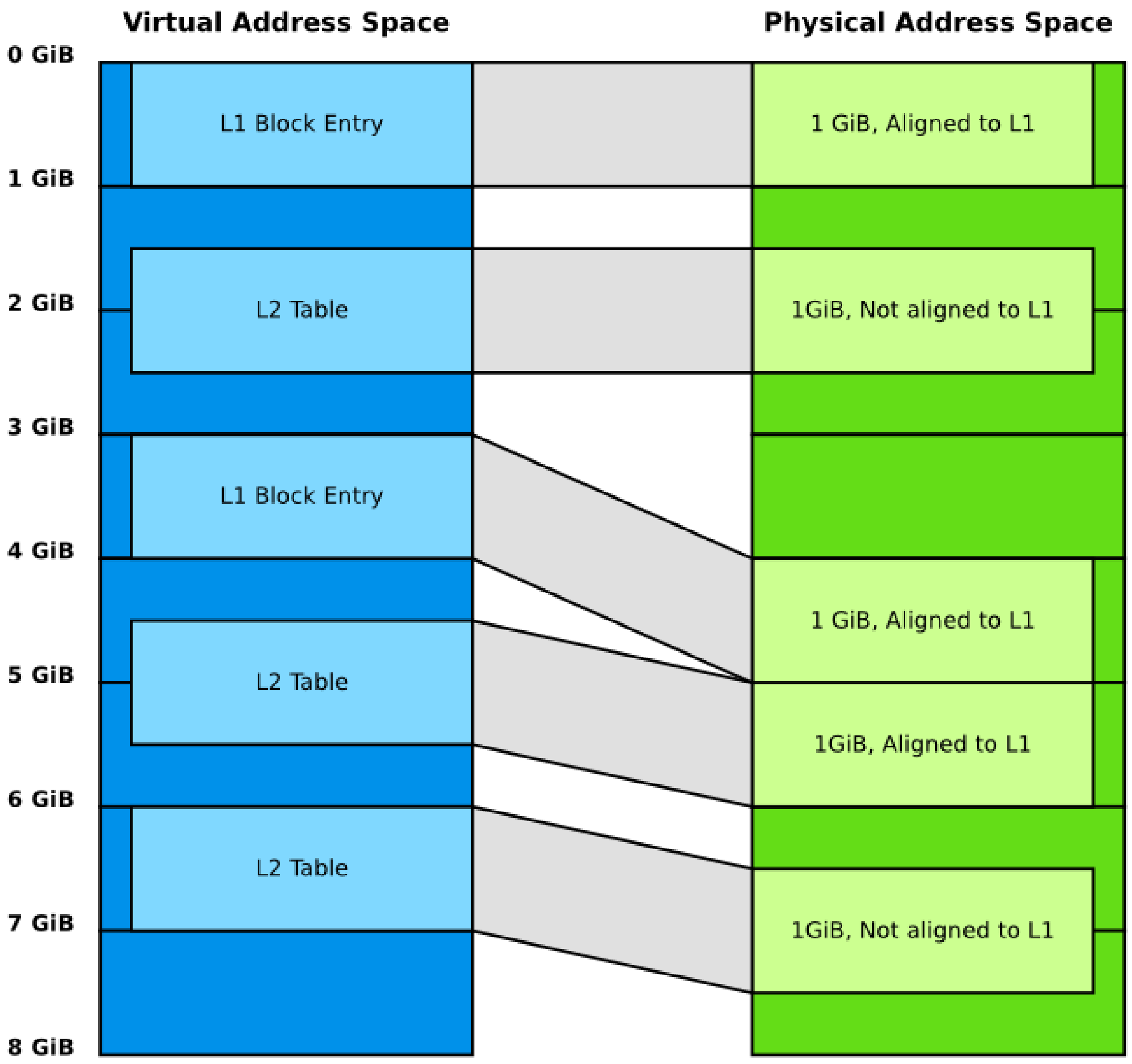
The mapping function is implemented as a recursive algorithm. It is however bound by the level of depth of the translation tables (the Armv8-A architecture allows up to 4 lookup levels).

By default[1], the algorithm will attempt to minimize the number of translation tables created to satisfy the user's request. It will favour mapping a region using the biggest possible blocks, only creating a sub-table if it is strictly necessary. This is to reduce the memory footprint of the firmware.

The most common reason for needing a sub-table is when a specific mapping requires a finer granularity. Misaligned regions also require a finer granularity than what the user may had originally expected, using a lot more memory than expected. The reason is that all levels of translation are restricted to address translations of the same granularity as the size of the blocks of that level. For example, for a 4 KiB page size, a level 2 block entry can only translate up to a granularity of 2 MiB. If the Physical Address is not aligned to 2 MiB then additional level 3 tables are also needed.

Note that not every translation level allows any type of descriptor. Depending on the page size, levels 0 and 1 of translation may only allow table descriptors. If a block entry could be able to describe a translation, but that level does not allow block descriptors, a table descriptor will have to be used instead, as well as additional tables at the next level.

[1] That is, when mmap regions do not enforce their mapping granularity.



The mmap regions are sorted in a way that simplifies the code that maps them. Even though this ordering is only strictly needed for overlapping static regions, it must also be applied for dynamic regions to maintain a consistent order of all regions at all times. As each new region is mapped, existing entries in the translation tables are checked to ensure consistency. Please refer to the comments in the source code of the core module for more details about the sorting algorithm in use.

TLB maintenance operations

The library takes care of performing TLB maintenance operations when required. For example, when the user requests removing a dynamic region, the library invalidates all TLB entries associated to that region to ensure that these changes are visible to subsequent execution, including speculative execution, that uses the changed translation table entries.

A counter-example is the initialization of translation tables. In this case, explicit TLB maintenance is not required. The Armv8-A architecture guarantees that all TLBs are disabled from reset and their contents have no effect on address translation at reset[2]. Therefore, the TLBs invalidation is deferred to the `enable_mmu*()` family of functions, just before the MMU is turned on.

[2] See section D4.9 `Translation Lookaside Buffers (TLBs)`, subsection `TLB behavior at reset` in Armv8-A, rev C.a.

TLB invalidation is not required when adding dynamic regions either. Dynamic regions are not allowed to overlap existing memory region. Therefore, if the dynamic mapping request is deemed legitimate, it automatically concerns memory that was not mapped in this translation regime and the library will have initialized its corresponding translation table entry to an invalid descriptor. Given that the TLBs are not architecturally permitted to hold any invalid translation table entry[3], this means that this mapping cannot be cached in the TLBs.

4.16 Chain of trust bindings

The device tree allows to describe the chain of trust with the help of 'cot' node which contain 'manifests' and 'images' as sub-nodes. 'manifests' and 'images' nodes contains number of sub-nodes (i.e. 'certificate' and 'image' nodes) mentioning properties of the certificate and image respectively.

Also, device tree describes 'non-volatile-counters' node which contains number of sub-nodes mentioning properties of all non-volatile-counters used in the chain of trust.

4.16.1 cot

This is root node which contains 'manifests' and 'images' as sub-nodes

4.16.2 Manifests and Certificate node bindings definition

- **Manifests node** Description: Container of certificate nodes.

  PROPERTIES

  - **compatible:** Usage: required

    Value type: <string>

    Definition: must be "arm, cert-descs"

- **Certificate node** Description:

  Describes certificate properties which are used during the authentication process.

  PROPERTIES

  - **root-certificate** Usage:

    Required for the certificate with no parent. In other words, certificates which are validated using root of trust public key.

    Value type: <boolean>

  - **image-id** Usage: Required for every certificate with unique id.

    Value type: <u32>

  - **parent** Usage:

    It refers to their parent image, which typically contains information to authenticate the certificate. This property is required for all non-root certificates.

[3] See section D4.10.1 `General TLB maintenance requirements` in Armv8-A, rev C.a.

This property is not required for root-certificates as root-certificates are validated using root of trust public key provided by platform.

Value type: <phandle>

- **signing-key** Usage:

  This property is used to refer public key node present in parent certificate node and it is required property for all non-root certificates which are authenticated using public-key present in parent certificate.

  This property is not required for root-certificates as root-certificates are validated using root of trust public key provided by platform.

  Value type: <phandle>

- **antirollback-counter** Usage:

  This property is used by all certificates which are protected against rollback attacks using a non-volatile counter and it is an optional property.

  This property is used to refer one of the non-volatile counter sub-node present in 'non-volatile counters' node.

  Value type: <phandle>

SUBNODES

- Description:

  Hash and public key information present in the certificate are shown by these nodes.

- **public key node** Description: Provide public key information in the certificate.

  PROPERTIES

  * **oid** Usage:

    This property provides the Object ID of public key provided in the certificate which the help of which public key information can be extracted.

    Value type: <string>

- **hash node** Description: Provide the hash information in the certificate.

  PROPERTIES

  * **oid** Usage:

    This property provides the Object ID of hash provided in the certificate which the help of which hash information can be extracted.

    Value type: <string>

Example:

```
cot {
   manifests {
      compatible = "arm, cert-descs”

      trusted-key-cert: trusted-key-cert {
         root-certificate;
         image-id = <TRUSTED_KEY_CERT_ID>;
         antirollback-counter = <&trusted_nv_counter>;

         trusted-world-pk: trusted-world-pk {
```

(continues on next page)

(continued from previous page)

```
            oid = TRUSTED_WORLD_PK_OID;
         };
         non-trusted-world-pk: non-trusted-world-pk {
            oid = NON_TRUSTED_WORLD_PK_OID;
         };
      };

      scp_fw_key_cert: scp_fw_key_cert {
         image-id = <SCP_FW_KEY_CERT_ID>;
         parent = <&trusted-key-cert>;
         signing-key = <&trusted_world_pk>;
         antirollback-counter = <&trusted_nv_counter>;

         scp_fw_content_pk: scp_fw_content_pk {
            oid = SCP_FW_CONTENT_CERT_PK_OID;
         };
      };
      .
      .
      .

      next-certificate {

      };
   };
};
```

4.16.3 Images and Image node bindings definition

- **Images node** Description: Container of image nodes

  PROPERTIES

  - **compatible:** Usage: required

    Value type: <string>

    Definition: must be "arm, img-descs"

- **Image node** Description:

  Describes image properties which will be used during authentication process.

  PROPERTIES

  - **image-id** Usage: Required for every image with unique id.

    Value type: <u32>

  - **parent** Usage:

    Required for every image to provide a reference to its parent image, which contains the necessary information to authenticate it.

    Value type: <phandle>

  - **hash** Usage:

    Required for all images which are validated using hash method. This property is used to refer hash node present in parent certificate node.

Value type: <phandle>

Note:

Currently, all images are validated using 'hash' method. In future, there may be multiple methods can be used to validate the image.

Example:

```
cot {
   images {
      compatible = "arm, img-descs";

      scp_bl2_image {
         image-id = <SCP_BL2_IMAGE_ID>;
         parent = <&scp_fw_content_cert>;
         hash = <&scp_fw_hash>;
      };

      .
      .
      .

      next-img {

      };
   };
};
```

4.16.4 non-volatile counter node binding definition

- **non-volatile counters node** Description: Contains properties for non-volatile counters.

  PROPERTIES

  - **compatible:** Usage: required

    Value type: <string>

    Definition: must be "arm, non-volatile-counter"

  - **#address-cells** Usage: required

    Value type: <u32>

    Definition:

    Must be set according to address size of non-volatile counter register

  - **#size-cells** Usage: required

    Value type: <u32>

    Definition: must be set to 0

  SUBNODE

  - **counters node** Description: Contains various non-volatile counters present in the platform.

    PROPERTIES

    - **id** Usage: Required for every nv-counter with unique id.

      Value type: <u32>

- **reg** Usage:

  Register base address of non-volatile counter and it is required property.

  Value type: <u32>

- **oid** Usage:

  This property provides the Object ID of non-volatile counter provided in the certificate and it is required property.

  Value type: <string>

Example: Below is non-volatile counters example for ARM platform

```
non_volatile_counters: non_volatile_counters {
     compatible = "arm, non-volatile-counter";
     #address-cells = <1>;
     #size-cells = <0>;

     trusted-nv-counter: trusted_nv_counter {
        id  = <TRUSTED_NV_CTR_ID>;
        reg = <TFW_NVCTR_BASE>;
        oid = TRUSTED_FW_NVCOUNTER_OID;
     };

     non_trusted_nv_counter: non_trusted_nv_counter {
        id  = <NON_TRUSTED_NV_CTR_ID>;
        reg = <NTFW_CTR_BASE>;
        oid = NON_TRUSTED_FW_NVCOUNTER_OID;
     };
};
```

4.16.5 Future update to chain of trust binding

This binding document needs to be revisited to generalise some terminologies which are currently specific to X.509 certificates for e.g. Object IDs.

CHAPTER

FIVE

SYSTEM DESIGN

5.1 Alternative Boot Flows

5.1.1 EL3 payloads alternative boot flow

On a pre-production system, the ability to execute arbitrary, bare-metal code at the highest exception level is required. It allows full, direct access to the hardware, for example to run silicon soak tests.

Although it is possible to implement some baremetal secure firmware from scratch, this is a complex task on some platforms, depending on the level of configuration required to put the system in the expected state.

Rather than booting a baremetal application, a possible compromise is to boot `EL3 payloads` through TF-A instead. This is implemented as an alternative boot flow, where a modified BL2 boots an EL3 payload, instead of loading the other BL images and passing control to BL31. It reduces the complexity of developing EL3 baremetal code by:

- putting the system into a known architectural state;
- taking care of platform secure world initialization;
- loading the SCP_BL2 image if required by the platform.

When booting an EL3 payload on Arm standard platforms, the configuration of the TrustZone controller is simplified such that only region 0 is enabled and is configured to permit secure access only. This gives full access to the whole DRAM to the EL3 payload.

The system is left in the same state as when entering BL31 in the default boot flow. In particular:

- Running in EL3;
- Current state is AArch64;
- Little-endian data access;
- All exceptions disabled;
- MMU disabled;
- Caches disabled.

Booting an EL3 payload

The EL3 payload image is a standalone image and is not part of the FIP. It is not loaded by TF-A. Therefore, there are 2 possible scenarios:

- The EL3 payload may reside in non-volatile memory (NVM) and execute in place. In this case, booting it is just a matter of specifying the right address in NVM through `EL3_PAYLOAD_BASE` when building TF-A.
- The EL3 payload needs to be loaded in volatile memory (e.g. DRAM) at run-time.

To help in the latter scenario, the `SPIN_ON_BL1_EXIT=1` build option can be used. The infinite loop that it introduces in BL1 stops execution at the right moment for a debugger to take control of the target and load the payload (for example, over JTAG).

It is expected that this loading method will work in most cases, as a debugger connection is usually available in a pre-production system. The user is free to use any other platform-specific mechanism to load the EL3 payload, though.

5.1.2 Preloaded BL33 alternative boot flow

Some platforms have the ability to preload BL33 into memory instead of relying on TF-A to load it. This may simplify packaging of the normal world code and improve performance in a development environment. When secure world cold boot is complete, TF-A simply jumps to a BL33 base address provided at build time.

For this option to be used, the `PRELOADED_BL33_BASE` build option has to be used when compiling TF-A. For example, the following command will create a FIP without a BL33 and prepare to jump to a BL33 image loaded at address 0x80000000:

```
make PRELOADED_BL33_BASE=0x80000000 PLAT=fvp all fip
```

5.2 Authentication Framework & Chain of Trust

The aim of this document is to describe the authentication framework implemented in Trusted Firmware-A (TF-A). This framework fulfills the following requirements:

1. It should be possible for a platform port to specify the Chain of Trust in terms of certificate hierarchy and the mechanisms used to verify a particular image/certificate.
2. The framework should distinguish between:
   - The mechanism used to encode and transport information, e.g. DER encoded X.509v3 certificates to ferry Subject Public Keys, hashes and non-volatile counters.
   - The mechanism used to verify the transported information i.e. the cryptographic libraries.

The framework has been designed following a modular approach illustrated in the next diagram:

```
+---------------+---------------+------------+
| Trusted       | Trusted       | Trusted    |
| Firmware      | Firmware      | Firmware   |
| Generic       | IO Framework  | Platform   |
| Code i.e.     | (IO)          | Port       |
| BL1/BL2 (GEN) |               | (PP)       |
```

(continues on next page)

(continued from previous page)

```
+----------------+----------------+-------------+
       ^                ^                ^
       |                |                |
       v                v                v
 +------------+    +------------+    +------------+
 |            |    |            |    | Image      |
 | Crypto     |    | Auth       |    | Parser     |
 | Module     |<->| Module     |<->| Module     |
 | (CM)       |    | (AM)       |    | (IPM)      |
 |            |    |            |    |            |
 +------------+    +------------+    +------------+
       ^                                 ^
       |                                 |
       v                                 v
+-----------------+             +------------------+
| Cryptographic   |             | Image Parser     |
| Libraries (CL)  |             | Libraries (IPL)  |
+-----------------+             +------------------+
              |                 |
              |                 |
              |                 |
              v                 v
             +------------------+
             | Misc. Libs e.g. |
             | ASN.1 decoder   |
             |                  |
             +------------------+

DIAGRAM 1.
```

This document describes the inner details of the authentication framework and the abstraction mechanisms available to specify a Chain of Trust.

5.2.1 Framework design

This section describes some aspects of the framework design and the rationale behind them. These aspects are key to verify a Chain of Trust.

Chain of Trust

A CoT is basically a sequence of authentication images which usually starts with a root of trust and culminates in a single data image. The following diagram illustrates how this maps to a CoT for the BL31 image described in the TBBR-Client specification.

```
+-------------------+       +--------------------+
| ROTPK/ROTPK Hash |------>| Trusted Key        |
+-------------------+       | Certificate        |
                            | (Auth Image)       |
                           /+--------------------+
                          /            |
                         /             |
                        /              |
                       /               |
                      L                v
```

(continues on next page)

(continued from previous page)

```
+------------------+       +-------------------+
| Trusted World    |------>| BL31 Key          |
| Public Key       |       | Certificate       |
+------------------+       | (Auth Image)      |
                           +-------------------+
                          /           |
                         /            |
                        /             |
                       /              |
                      /               v
+------------------+ L     +-------------------+
| BL31 Content     |------>| BL31 Content      |
| Certificate PK   |       | Certificate       |
+------------------+       | (Auth Image)      |
                           +-------------------+
                          /           |
                         /            |
                        /             |
                       /              |
                      /               v
+------------------+ L     +-------------------+
| BL31 Hash        |------>| BL31 Image        |
|                  |       | (Data Image)      |
+------------------+       |                   |
                           +-------------------+

DIAGRAM 2.
```

The root of trust is usually a public key (ROTPK) that has been burnt in the platform and cannot be modified.

Image types

Images in a CoT are categorised as authentication and data images. An authentication image contains information to authenticate a data image or another authentication image. A data image is usually a boot loader binary, but it could be any other data that requires authentication.

Component responsibilities

For every image in a Chain of Trust, the following high level operations are performed to verify it:

1. Allocate memory for the image either statically or at runtime.
2. Identify the image and load it in the allocated memory.
3. Check the integrity of the image as per its type.
4. Authenticate the image as per the cryptographic algorithms used.
5. If the image is an authentication image, extract the information that will be used to authenticate the next image in the CoT.

In Diagram 1, each component is responsible for one or more of these operations. The responsibilities are briefly described below.

TF-A Generic code and IO framework (GEN/IO)

These components are responsible for initiating the authentication process for a particular image in BL1 or BL2. For each BL image that requires authentication, the Generic code asks recursively the Authentication module what is the parent image until either an authenticated image or the ROT is reached. Then the Generic code calls the IO framework to load the image and calls the Authentication module to authenticate it, following the CoT from ROT to Image.

TF-A Platform Port (PP)

The platform is responsible for:

1. Specifying the CoT for each image that needs to be authenticated. Details of how a CoT can be specified by the platform are explained later. The platform also specifies the authentication methods and the parsing method used for each image.
2. Statically allocating memory for each parameter in each image which is used for verifying the CoT, e.g. memory for public keys, hashes etc.
3. Providing the ROTPK or a hash of it.
4. Providing additional information to the IPM to enable it to identify and extract authentication parameters contained in an image, e.g. if the parameters are stored as X509v3 extensions, the corresponding OID must be provided.
5. Fulfill any other memory requirements of the IPM and the CM (not currently described in this document).
6. Export functions to verify an image which uses an authentication method that cannot be interpreted by the CM, e.g. if an image has to be verified using a NV counter, then the value of the counter to compare with can only be provided by the platform.
7. Export a custom IPM if a proprietary image format is being used (described later).

Authentication Module (AM)

It is responsible for:

1. Providing the necessary abstraction mechanisms to describe a CoT. Amongst other things, the authentication and image parsing methods must be specified by the PP in the CoT.
2. Verifying the CoT passed by GEN by utilising functionality exported by the PP, IPM and CM.
3. Tracking which images have been verified. In case an image is a part of multiple CoTs then it should be verified only once e.g. the Trusted World Key Certificate in the TBBR-Client spec. contains information to verify SCP_BL2, BL31, BL32 each of which have a separate CoT. (This responsibility has not been described in this document but should be trivial to implement).
4. Reusing memory meant for a data image to verify authentication images e.g. in the CoT described in Diagram 2, each certificate can be loaded and verified in the memory reserved by the platform for the BL31 image. By the time BL31 (the data image) is loaded, all information to authenticate it will have been extracted from the parent image i.e. BL31 content certificate. It is assumed that the size of an authentication image will never exceed the size of a data image. It should be possible to verify this at build time using asserts.

Cryptographic Module (CM)

The CM is responsible for providing an API to:

1. Verify a digital signature.
2. Verify a hash.

The CM does not include any cryptography related code, but it relies on an external library to perform the cryptographic operations. A Crypto-Library (CL) linking the CM and the external library must be implemented. The following functions must be provided by the CL:

```
void (*init)(void);
int (*verify_signature)(void *data_ptr, unsigned int data_len,
                        void *sig_ptr, unsigned int sig_len,
                        void *sig_alg, unsigned int sig_alg_len,
                        void *pk_ptr, unsigned int pk_len);
int (*verify_hash)(void *data_ptr, unsigned int data_len,
                   void *digest_info_ptr, unsigned int digest_info_len);
```

These functions are registered in the CM using the macro:

```
REGISTER_CRYPTO_LIB(_name, _init, _verify_signature, _verify_hash);
```

`_name` must be a string containing the name of the CL. This name is used for debugging purposes.

Image Parser Module (IPM)

The IPM is responsible for:

1. Checking the integrity of each image loaded by the IO framework.
2. Extracting parameters used for authenticating an image based upon a description provided by the platform in the CoT descriptor.

Images may have different formats (for example, authentication images could be x509v3 certificates, signed ELF files or any other platform specific format). The IPM allows to register an Image Parser Library (IPL) for every image format used in the CoT. This library must implement the specific methods to parse the image. The IPM obtains the image format from the CoT and calls the right IPL to check the image integrity and extract the authentication parameters.

See Section "Describing the image parsing methods" for more details about the mechanism the IPM provides to define and register IPLs.

Authentication methods

The AM supports the following authentication methods:

1. Hash
2. Digital signature

The platform may specify these methods in the CoT in case it decides to define a custom CoT instead of reusing a predefined one.

If a data image uses multiple methods, then all the methods must be a part of the same CoT. The number and type of parameters are method specific. These parameters should be obtained from the parent image using the IPM.

1. Hash

   Parameters:

   1. A pointer to data to hash
   2. Length of the data
   3. A pointer to the hash
   4. Length of the hash

   The hash will be represented by the DER encoding of the following ASN.1 type:

   ```
   DigestInfo ::= SEQUENCE {
       digestAlgorithm  DigestAlgorithmIdentifier,
       digest           Digest
   }
   ```

   This ASN.1 structure makes it possible to remove any assumption about the type of hash algorithm used as this information accompanies the hash. This should allow the Cryptography Library (CL) to support multiple hash algorithm implementations.

2. Digital Signature

   Parameters:

   1. A pointer to data to sign
   2. Length of the data
   3. Public Key Algorithm
   4. Public Key value
   5. Digital Signature Algorithm
   6. Digital Signature value

   The Public Key parameters will be represented by the DER encoding of the following ASN.1 type:

   ```
   SubjectPublicKeyInfo  ::=  SEQUENCE  {
       algorithm         AlgorithmIdentifier{PUBLIC-KEY,{PublicKeyAlgorithms}},
       subjectPublicKey  BIT STRING  }
   ```

   The Digital Signature Algorithm will be represented by the DER encoding of the following ASN.1 types.

   ```
   AlgorithmIdentifier {ALGORITHM:IOSet } ::= SEQUENCE {
       algorithm         ALGORITHM.&id({IOSet}),
       parameters        ALGORITHM.&Type({IOSet}{@algorithm}) OPTIONAL
   }
   ```

   The digital signature will be represented by:

   ```
   signature  ::=  BIT STRING
   ```

The authentication framework will use the image descriptor to extract all the information related to authentication.

5.2.2 Specifying a Chain of Trust

A CoT can be described as a set of image descriptors linked together in a particular order. The order dictates the sequence in which they must be verified. Each image has a set of properties which allow the AM to verify it. These properties are described below.

The PP is responsible for defining a single or multiple CoTs for a data image. Unless otherwise specified, the data structures described in the following sections are populated by the PP statically.

Describing the image parsing methods

The parsing method refers to the format of a particular image. For example, an authentication image that represents a certificate could be in the X.509v3 format. A data image that represents a boot loader stage could be in raw binary or ELF format. The IPM supports three parsing methods. An image has to use one of the three methods described below. An IPL is responsible for interpreting a single parsing method. There has to be one IPL for every method used by the platform.

1. Raw format: This format is effectively a nop as an image using this method is treated as being in raw binary format e.g. boot loader images used by TF-A. This method should only be used by data images.
2. X509V3 method: This method uses industry standards like X.509 to represent PKI certificates (authentication images). It is expected that open source libraries will be available which can be used to parse an image represented by this method. Such libraries can be used to write the corresponding IPL e.g. the X.509 parsing library code in mbed TLS.
3. Platform defined method: This method caters for platform specific proprietary standards to represent authentication or data images. For example, The signature of a data image could be appended to the data image raw binary. A header could be prepended to the combined blob to specify the extents of each component. The platform will have to implement the corresponding IPL to interpret such a format.

The following enum can be used to define these three methods.

```
typedef enum img_type_enum {
    IMG_RAW,            /* Binary image */
    IMG_PLAT,           /* Platform specific format */
    IMG_CERT,           /* X509v3 certificate */
    IMG_MAX_TYPES,
} img_type_t;
```

An IPL must provide functions with the following prototypes:

```
void init(void);
int check_integrity(void *img, unsigned int img_len);
int get_auth_param(const auth_param_type_desc_t *type_desc,
                      void *img, unsigned int img_len,
                      void **param, unsigned int *param_len);
```

An IPL for each type must be registered using the following macro:

```
REGISTER_IMG_PARSER_LIB(_type, _name, _init, _check_int, _get_param)
```

- `_type`: one of the types described above.
- `_name`: a string containing the IPL name for debugging purposes.
- `_init`: initialization function pointer.
- `_check_int`: check image integrity function pointer.

- _get_param: extract authentication parameter function pointer.

The init() function will be used to initialize the IPL.

The check_integrity() function is passed a pointer to the memory where the image has been loaded by the IO framework and the image length. It should ensure that the image is in the format corresponding to the parsing method and has not been tampered with. For example, RFC-2459 describes a validation sequence for an X.509 certificate.

The get_auth_param() function is passed a parameter descriptor containing information about the parameter (type_desc and cookie) to identify and extract the data corresponding to that parameter from an image. This data will be used to verify either the current or the next image in the CoT sequence.

Each image in the CoT will specify the parsing method it uses. This information will be used by the IPM to find the right parser descriptor for the image.

Describing the authentication method(s)

As part of the CoT, each image has to specify one or more authentication methods which will be used to verify it. As described in the Section "Authentication methods", there are three methods supported by the AM.

```
typedef enum {
    AUTH_METHOD_NONE,
    AUTH_METHOD_HASH,
    AUTH_METHOD_SIG,
    AUTH_METHOD_NUM
} auth_method_type_t;
```

The AM defines the type of each parameter used by an authentication method. It uses this information to:

1. Specify to the get_auth_param() function exported by the IPM, which parameter should be extracted from an image.
2. Correctly marshall the parameters while calling the verification function exported by the CM and PP.
3. Extract authentication parameters from a parent image in order to verify a child image e.g. to verify the certificate image, the public key has to be obtained from the parent image.

```
typedef enum {
    AUTH_PARAM_NONE,
    AUTH_PARAM_RAW_DATA,        /* Raw image data */
    AUTH_PARAM_SIG,         /* The image signature */
    AUTH_PARAM_SIG_ALG,     /* The image signature algorithm */
    AUTH_PARAM_HASH,        /* A hash (including the algorithm) */
    AUTH_PARAM_PUB_KEY,     /* A public key */
} auth_param_type_t;
```

The AM defines the following structure to identify an authentication parameter required to verify an image.

```
typedef struct auth_param_type_desc_s {
    auth_param_type_t type;
    void *cookie;
} auth_param_type_desc_t;
```

cookie is used by the platform to specify additional information to the IPM which enables it to uniquely identify the parameter that should be extracted from an image. For example, the hash of a BL3x image in its corresponding content certificate is stored in an X509v3 custom extension field. An extension field can only be identified using an OID. In this case, the cookie could contain the pointer to the OID defined by the platform for the hash extension field while the type field could be set to AUTH_PARAM_HASH. A value of 0 for the cookie field means that it is not used.

For each method, the AM defines a structure with the parameters required to verify the image.

```
/*
 * Parameters for authentication by hash matching
 */
typedef struct auth_method_param_hash_s {
    auth_param_type_desc_t *data;   /* Data to hash */
    auth_param_type_desc_t *hash;   /* Hash to match with */
} auth_method_param_hash_t;

/*
 * Parameters for authentication by signature
 */
typedef struct auth_method_param_sig_s {
    auth_param_type_desc_t *pk; /* Public key */
    auth_param_type_desc_t *sig;    /* Signature to check */
    auth_param_type_desc_t *alg;    /* Signature algorithm */
    auth_param_type_desc_t *tbs;    /* Data signed */
} auth_method_param_sig_t;
```

The AM defines the following structure to describe an authentication method for verifying an image

```
/*
 * Authentication method descriptor
 */
typedef struct auth_method_desc_s {
    auth_method_type_t type;
    union {
        auth_method_param_hash_t hash;
        auth_method_param_sig_t sig;
    } param;
} auth_method_desc_t;
```

Using the method type specified in the `type` field, the AM finds out what field needs to access within the `param` union.

Storing Authentication parameters

A parameter described by `auth_param_type_desc_t` to verify an image could be obtained from either the image itself or its parent image. The memory allocated for loading the parent image will be reused for loading the child image. Hence parameters which are obtained from the parent for verifying a child image need to have memory allocated for them separately where they can be stored. This memory must be statically allocated by the platform port.

The AM defines the following structure to store the data corresponding to an authentication parameter.

```
typedef struct auth_param_data_desc_s {
    void *auth_param_ptr;
    unsigned int auth_param_len;
} auth_param_data_desc_t;
```

The `auth_param_ptr` field is initialized by the platform. The `auth_param_len` field is used to specify the length of the data in the memory.

For parameters that can be obtained from the child image itself, the IPM is responsible for populating the `auth_param_ptr` and `auth_param_len` fields while executing the `img_get_auth_param()` function.

The AM defines the following structure to enable an image to describe the parameters that should be extracted from it and used to verify the next image (child) in a CoT.

```
typedef struct auth_param_desc_s {
    auth_param_type_desc_t type_desc;
    auth_param_data_desc_t data;
} auth_param_desc_t;
```

Describing an image in a CoT

An image in a CoT is a consolidation of the following aspects of a CoT described above.

1. A unique identifier specified by the platform which allows the IO framework to locate the image in a FIP and load it in the memory reserved for the data image in the CoT.
2. A parsing method which is used by the AM to find the appropriate IPM.
3. Authentication methods and their parameters as described in the previous section. These are used to verify the current image.
4. Parameters which are used to verify the next image in the current CoT. These parameters are specified only by authentication images and can be extracted from the current image once it has been verified.

The following data structure describes an image in a CoT.

```
typedef struct auth_img_desc_s {
    unsigned int img_id;
    const struct auth_img_desc_s *parent;
    img_type_t img_type;
    const auth_method_desc_t *const img_auth_methods;
    const auth_param_desc_t *const authenticated_data;
} auth_img_desc_t;
```

A CoT is defined as an array of pointers to `auth_image_desc_t` structures linked together by the `parent` field. Those nodes with no parent must be authenticated using the ROTPK stored in the platform.

5.2.3 Implementation example

This section is a detailed guide explaining a trusted boot implementation using the authentication framework. This example corresponds to the Applicative Functional Mode (AFM) as specified in the TBBR-Client document. It is recommended to read this guide along with the source code.

The TBBR CoT

CoT specific to BL1 and BL2 can be found in `drivers/auth/tbbr/tbbr_cot_bl1.c` and `drivers/auth/tbbr/tbbr_cot_bl2.c` respectively. The common CoT used across BL1 and BL2 can be found in `drivers/auth/tbbr/tbbr_cot_common.c`. This CoT consists of an array of pointers to image descriptors and it is registered in the framework using the macro `REGISTER_COT(cot_desc)`, where `cot_desc` must be the name of the array (passing a pointer or any other type of indirection will cause the registration process to fail).

The number of images participating in the boot process depends on the CoT. There is, however, a minimum set of images that are mandatory in TF-A and thus all CoTs must present:

- `BL2`
- `SCP_BL2` (platform specific)
- `BL31`
- `BL32` (optional)

- BL33

The TBBR specifies the additional certificates that must accompany these images for a proper authentication. Details about the TBBR CoT may be found in the *Trusted Board Boot* document.

Following the *Porting Guide*, a platform must provide unique identifiers for all the images and certificates that will be loaded during the boot process. If a platform is using the TBBR as a reference for trusted boot, these identifiers can be obtained from `include/common/tbbr/tbbr_img_def.h`. Arm platforms include this file in `include/plat/arm/common/arm_def.h`. Other platforms may also include this file or provide their own identifiers.

Important: the authentication module uses these identifiers to index the CoT array, so the descriptors location in the array must match the identifiers.

Each image descriptor must specify:

- `img_id`: the corresponding image unique identifier defined by the platform.
- `img_type`: the image parser module uses the image type to call the proper parsing library to check the image integrity and extract the required authentication parameters. Three types of images are currently supported:
  - `IMG_RAW`: image is a raw binary. No parsing functions are available, other than reading the whole image.
  - `IMG_PLAT`: image format is platform specific. The platform may use this type for custom images not directly supported by the authentication framework.
  - `IMG_CERT`: image is an x509v3 certificate.
- `parent`: pointer to the parent image descriptor. The parent will contain the information required to authenticate the current image. If the parent is NULL, the authentication parameters will be obtained from the platform (i.e. the BL2 and Trusted Key certificates are signed with the ROT private key, whose public part is stored in the platform).
- `img_auth_methods`: this points to an array which defines the authentication methods that must be checked to consider an image authenticated. Each method consists of a type and a list of parameter descriptors. A parameter descriptor consists of a type and a cookie which will point to specific information required to extract that parameter from the image (i.e. if the parameter is stored in an x509v3 extension, the cookie will point to the extension OID). Depending on the method type, a different number of parameters must be specified. This pointer should not be NULL. Supported methods are:
  - `AUTH_METHOD_HASH`: the hash of the image must match the hash extracted from the parent image. The following parameter descriptors must be specified:
    - `data`: data to be hashed (obtained from current image)
    - `hash`: reference hash (obtained from parent image)
  - `AUTH_METHOD_SIG`: the image (usually a certificate) must be signed with the private key whose public part is extracted from the parent image (or the platform if the parent is NULL). The following parameter descriptors must be specified:
    - `pk`: the public key (obtained from parent image)
    - `sig`: the digital signature (obtained from current image)
    - `alg`: the signature algorithm used (obtained from current image)
    - `data`: the data to be signed (obtained from current image)
- `authenticated_data`: this array pointer indicates what authentication parameters must be extracted from an image once it has been authenticated. Each parameter consists of a parameter descriptor and the buffer address/size to store the parameter. The CoT is responsible for allocating the required memory to store the parameters. This pointer may be NULL.

In the `tbbr_cot*.c` file, a set of buffers are allocated to store the parameters extracted from the certificates. In the case of the TBBR CoT, these parameters are hashes and public keys. In DER format, an RSA-4096 public key requires 550 bytes, and a hash requires 51 bytes. Depending on the CoT and the authentication process, some of the buffers may be reused at different stages during the boot.

Next in that file, the parameter descriptors are defined. These descriptors will be used to extract the parameter data from the corresponding image.

Example: the BL31 Chain of Trust

Four image descriptors form the BL31 Chain of Trust:

```
static const auth_img_desc_t trusted_key_cert = {
        .img_id = TRUSTED_KEY_CERT_ID,
        .img_type = IMG_CERT,
        .parent = NULL,
        .img_auth_methods =  (const auth_method_desc_t[AUTH_METHOD_NUM]) {
                [0] = {
                        .type = AUTH_METHOD_SIG,
                        .param.sig = {
                                .pk = &subject_pk,
                                .sig = &sig,
                                .alg = &sig_alg,
                                .data = &raw_data
                        }
                },
                [1] = {
                        .type = AUTH_METHOD_NV_CTR,
                        .param.nv_ctr = {
                                .cert_nv_ctr = &trusted_nv_ctr,
                                .plat_nv_ctr = &trusted_nv_ctr
                        }
                }
        },
        .authenticated_data = (const auth_param_desc_t[COT_MAX_VERIFIED_PARAMS]) {
                [0] = {
                        .type_desc = &trusted_world_pk,
                        .data = {
                                .ptr = (void *)trusted_world_pk_buf,
                                .len = (unsigned int)PK_DER_LEN
                        }
                },
                [1] = {
                        .type_desc = &non_trusted_world_pk,
                        .data = {
                                .ptr = (void *)non_trusted_world_pk_buf,
                                .len = (unsigned int)PK_DER_LEN
                        }
                }
        }
};
static const auth_img_desc_t soc_fw_key_cert = {
        .img_id = SOC_FW_KEY_CERT_ID,
        .img_type = IMG_CERT,
        .parent = &trusted_key_cert,
        .img_auth_methods =  (const auth_method_desc_t[AUTH_METHOD_NUM]) {
                [0] = {
```

(continues on next page)

(continued from previous page)

```
                        .type = AUTH_METHOD_SIG,
                        .param.sig = {
                                .pk = &trusted_world_pk,
                                .sig = &sig,
                                .alg = &sig_alg,
                                .data = &raw_data
                        }
                },
                [1] = {
                        .type = AUTH_METHOD_NV_CTR,
                        .param.nv_ctr = {
                                .cert_nv_ctr = &trusted_nv_ctr,
                                .plat_nv_ctr = &trusted_nv_ctr
                        }
                }
        },
        .authenticated_data = (const auth_param_desc_t[COT_MAX_VERIFIED_PARAMS]) {
                [0] = {
                        .type_desc = &soc_fw_content_pk,
                        .data = {
                                .ptr = (void *)content_pk_buf,
                                .len = (unsigned int)PK_DER_LEN
                        }
                }
        }
};
static const auth_img_desc_t soc_fw_content_cert = {
        .img_id = SOC_FW_CONTENT_CERT_ID,
        .img_type = IMG_CERT,
        .parent = &soc_fw_key_cert,
        .img_auth_methods =  (const auth_method_desc_t[AUTH_METHOD_NUM]) {
                [0] = {
                        .type = AUTH_METHOD_SIG,
                        .param.sig = {
                                .pk = &soc_fw_content_pk,
                                .sig = &sig,
                                .alg = &sig_alg,
                                .data = &raw_data
                        }
                },
                [1] = {
                        .type = AUTH_METHOD_NV_CTR,
                        .param.nv_ctr = {
                                .cert_nv_ctr = &trusted_nv_ctr,
                                .plat_nv_ctr = &trusted_nv_ctr
                        }
                }
        },
        .authenticated_data = (const auth_param_desc_t[COT_MAX_VERIFIED_PARAMS]) {
                [0] = {
                        .type_desc = &soc_fw_hash,
                        .data = {
                                .ptr = (void *)soc_fw_hash_buf,
                                .len = (unsigned int)HASH_DER_LEN
                        }
                },
                [1] = {
```

(continues on next page)

(continued from previous page)

```
                        .type_desc = &soc_fw_config_hash,
                        .data = {
                                .ptr = (void *)soc_fw_config_hash_buf,
                                .len = (unsigned int)HASH_DER_LEN
                        }
                }
        }
};
static const auth_img_desc_t bl31_image = {
        .img_id = BL31_IMAGE_ID,
        .img_type = IMG_RAW,
        .parent = &soc_fw_content_cert,
        .img_auth_methods =  (const auth_method_desc_t[AUTH_METHOD_NUM]) {
                [0] = {
                        .type = AUTH_METHOD_HASH,
                        .param.hash = {
                                .data = &raw_data,
                                .hash = &soc_fw_hash
                        }
                }
        }
};
```

The **Trusted Key certificate** is signed with the ROT private key and contains the Trusted World public key and the Non-Trusted World public key as x509v3 extensions. This must be specified in the image descriptor using the `img_auth_methods` and `authenticated_data` arrays, respectively.

The Trusted Key certificate is authenticated by checking its digital signature using the ROTPK. Four parameters are required to check a signature: the public key, the algorithm, the signature and the data that has been signed. Therefore, four parameter descriptors must be specified with the authentication method:

- `subject_pk`: parameter descriptor of type `AUTH_PARAM_PUB_KEY`. This type is used to extract a public key from the parent image. If the cookie is an OID, the key is extracted from the corresponding x509v3 extension. If the cookie is NULL, the subject public key is retrieved. In this case, because the parent image is NULL, the public key is obtained from the platform (this key will be the ROTPK).
- `sig`: parameter descriptor of type `AUTH_PARAM_SIG`. It is used to extract the signature from the certificate.
- `sig_alg`: parameter descriptor of type `AUTH_PARAM_SIG`. It is used to extract the signature algorithm from the certificate.
- `raw_data`: parameter descriptor of type `AUTH_PARAM_RAW_DATA`. It is used to extract the data to be signed from the certificate.

Once the signature has been checked and the certificate authenticated, the Trusted World public key needs to be extracted from the certificate. A new entry is created in the `authenticated_data` array for that purpose. In that entry, the corresponding parameter descriptor must be specified along with the buffer address to store the parameter value. In this case, the `trusted_world_pk` descriptor is used to extract the public key from an x509v3 extension with OID `TRUSTED_WORLD_PK_OID`. The BL31 key certificate will use this descriptor as parameter in the signature authentication method. The key is stored in the `trusted_world_pk_buf` buffer.

The **BL31 Key certificate** is authenticated by checking its digital signature using the Trusted World public key obtained previously from the Trusted Key certificate. In the image descriptor, we specify a single authentication method by signature whose public key is the `trusted_world_pk`. Once this certificate has been authenticated, we have to extract the BL31 public key, stored in the extension specified by `soc_fw_content_pk`. This key will be copied to the `content_pk_buf` buffer.

The **BL31 certificate** is authenticated by checking its digital signature using the BL31 public key obtained previously from the BL31 Key certificate. We specify the authentication method using `soc_fw_content_pk` as public key. After authentication, we need to extract the BL31 hash, stored in the extension specified by `soc_fw_hash`. This hash will be copied to the `soc_fw_hash_buf` buffer.

The **BL31 image** is authenticated by calculating its hash and matching it with the hash obtained from the BL31 certificate. The image descriptor contains a single authentication method by hash. The parameters to the hash method are the reference hash, `soc_fw_hash`, and the data to be hashed. In this case, it is the whole image, so we specify `raw_data`.

The image parser library

The image parser module relies on libraries to check the image integrity and extract the authentication parameters. The number and type of parser libraries depend on the images used in the CoT. Raw images do not need a library, so only an x509v3 library is required for the TBBR CoT.

Arm platforms will use an x509v3 library based on mbed TLS. This library may be found in `drivers/auth/mbedtls/mbedtls_x509_parser.c`. It exports three functions:

```
void init(void);
int check_integrity(void *img, unsigned int img_len);
int get_auth_param(const auth_param_type_desc_t *type_desc,
                   void *img, unsigned int img_len,
                   void **param, unsigned int *param_len);
```

The library is registered in the framework using the macro `REGISTER_IMG_PARSER_LIB()`. Each time the image parser module needs to access an image of type `IMG_CERT`, it will call the corresponding function exported in this file.

The build system must be updated to include the corresponding library and mbed TLS sources. Arm platforms use the `arm_common.mk` file to pull the sources.

The cryptographic library

The cryptographic module relies on a library to perform the required operations, i.e. verify a hash or a digital signature. Arm platforms will use a library based on mbed TLS, which can be found in `drivers/auth/mbedtls/mbedtls_crypto.c`. This library is registered in the authentication framework using the macro `REGISTER_CRYPTO_LIB()` and exports four functions:

```
void init(void);
int verify_signature(void *data_ptr, unsigned int data_len,
                     void *sig_ptr, unsigned int sig_len,
                     void *sig_alg, unsigned int sig_alg_len,
                     void *pk_ptr, unsigned int pk_len);
int verify_hash(void *data_ptr, unsigned int data_len,
                void *digest_info_ptr, unsigned int digest_info_len);
int auth_decrypt(enum crypto_dec_algo dec_algo, void *data_ptr,
                 size_t len, const void *key, unsigned int key_len,
                 unsigned int key_flags, const void *iv,
                 unsigned int iv_len, const void *tag,
                 unsigned int tag_len)
```

The mbedTLS library algorithm support is configured by both the `TF_MBEDTLS_KEY_ALG` and `TF_MBEDTLS_KEY_SIZE` variables.

- `TF_MBEDTLS_KEY_ALG` can take in 3 values: *rsa*, *ecdsa* or *rsa+ecdsa*. This variable allows the Makefile to include the corresponding sources in the build for the various algorithms. Setting the variable to *rsa+ecdsa* enables support for both rsa and ecdsa algorithms in the mbedTLS library.

- `TF_MBEDTLS_KEY_SIZE` sets the supported RSA key size for TFA. Valid values include 1024, 2048, 3072 and 4096.
- `TF_MBEDTLS_USE_AES_GCM` enables the authenticated decryption support based on AES-GCM algorithm. Valid values are 0 and 1.

Note: If code size is a concern, the build option `MBEDTLS_SHA256_SMALLER` can be defined in the platform Makefile. It will make mbed TLS use an implementation of SHA-256 with smaller memory footprint (~1.5 KB less) but slower (~30%).

5.3 Arm CPU Specific Build Macros

This document describes the various build options present in the CPU specific operations framework to enable errata workarounds and to enable optimizations for a specific CPU on a platform.

5.3.1 Security Vulnerability Workarounds

TF-A exports a series of build flags which control which security vulnerability workarounds should be applied at runtime.

- `WORKAROUND_CVE_2017_5715`: Enables the security workaround for CVE-2017-5715. This flag can be set to 0 by the platform if none of the PEs in the system need the workaround. Setting this flag to 0 provides no performance benefit for non-affected platforms, it just helps to comply with the recommendation in the spec regarding workaround discovery. Defaults to 1.
- `WORKAROUND_CVE_2018_3639`: Enables the security workaround for CVE-2018-3639. Defaults to 1. The TF-A project recommends to keep the default value of 1 even on platforms that are unaffected by CVE-2018-3639, in order to comply with the recommendation in the spec regarding workaround discovery.
- `DYNAMIC_WORKAROUND_CVE_2018_3639`: Enables dynamic mitigation for CVE-2018-3639. This build option should be set to 1 if the target platform contains at least 1 CPU that requires dynamic mitigation. Defaults to 0.

5.3.2 CPU Errata Workarounds

TF-A exports a series of build flags which control the errata workarounds that are applied to each CPU by the reset handler. The errata details can be found in the CPU specific errata documents published by Arm:

- Cortex-A53 MPCore Software Developers Errata Notice
- Cortex-A57 MPCore Software Developers Errata Notice
- Cortex-A72 MPCore Software Developers Errata Notice

The errata workarounds are implemented for a particular revision or a set of processor revisions. This is checked by the reset handler at runtime. Each errata workaround is identified by its `ID` as specified in the processor's errata notice document. The format of the define used to enable/disable the errata workaround is `ERRATA_<Processor name>_<ID>`, where the `Processor name` is for example `A57` for the `Cortex_A57` CPU.

Refer to *CPU errata status reporting* for information on how to write errata workaround functions.

All workarounds are disabled by default. The platform is responsible for enabling these workarounds according to its requirement by defining the errata workaround build flags in the platform specific makefile. In case these workarounds are enabled for the wrong CPU revision then the errata workaround is not applied. In the DEBUG build, this is indicated by printing a warning to the crash console.

In the current implementation, a platform which has more than 1 variant with different revisions of a processor has no runtime mechanism available for it to specify which errata workarounds should be enabled or not.

The value of the build flags is 0 by default, that is, disabled. A value of 1 will enable it.

For Cortex-A9, the following errata build flags are defined :

- `ERRATA_A9_794073`: This applies errata 794073 workaround to Cortex-A9 CPU. This needs to be enabled for all revisions of the CPU.

For Cortex-A15, the following errata build flags are defined :

- `ERRATA_A15_816470`: This applies errata 816470 workaround to Cortex-A15 CPU. This needs to be enabled only for revision >= r3p0 of the CPU.
- `ERRATA_A15_827671`: This applies errata 827671 workaround to Cortex-A15 CPU. This needs to be enabled only for revision >= r3p0 of the CPU.

For Cortex-A17, the following errata build flags are defined :

- `ERRATA_A17_852421`: This applies errata 852421 workaround to Cortex-A17 CPU. This needs to be enabled only for revision <= r1p2 of the CPU.
- `ERRATA_A17_852423`: This applies errata 852423 workaround to Cortex-A17 CPU. This needs to be enabled only for revision <= r1p2 of the CPU.

For Cortex-A35, the following errata build flags are defined :

- `ERRATA_A35_855472`: This applies errata 855472 workaround to Cortex-A35 CPUs. This needs to be enabled only for revision r0p0 of Cortex-A35.

For Cortex-A53, the following errata build flags are defined :

- `ERRATA_A53_819472`: This applies errata 819472 workaround to all CPUs. This needs to be enabled only for revision <= r0p1 of Cortex-A53.
- `ERRATA_A53_824069`: This applies errata 824069 workaround to all CPUs. This needs to be enabled only for revision <= r0p2 of Cortex-A53.
- `ERRATA_A53_826319`: This applies errata 826319 workaround to Cortex-A53 CPU. This needs to be enabled only for revision <= r0p2 of the CPU.
- `ERRATA_A53_827319`: This applies errata 827319 workaround to all CPUs. This needs to be enabled only for revision <= r0p2 of Cortex-A53.
- `ERRATA_A53_835769`: This applies erratum 835769 workaround at compile and link time to Cortex-A53 CPU. This needs to be enabled for some variants of revision <= r0p4. This workaround can lead the linker to create `*.stub` sections.
- `ERRATA_A53_836870`: This applies errata 836870 workaround to Cortex-A53 CPU. This needs to be enabled only for revision <= r0p3 of the CPU. From r0p4 and onwards, this errata is enabled by default in hardware.
- `ERRATA_A53_843419`: This applies erratum 843419 workaround at link time to Cortex-A53 CPU. This needs to be enabled for some variants of revision <= r0p4. This workaround can lead the linker to emit `*.stub` sections which are 4kB aligned.
- `ERRATA_A53_855873`: This applies errata 855873 workaround to Cortex-A53 CPUs. Though the erratum is present in every revision of the CPU, this workaround is only applied to CPUs from r0p3 onwards, which

feature a chicken bit in CPUACTLR_EL1 to enable a hardware workaround. Earlier revisions of the CPU have other errata which require the same workaround in software, so they should be covered anyway.

- `ERRATA_A53_1530924`: This applies errata 1530924 workaround to all revisions of Cortex-A53 CPU.

For Cortex-A55, the following errata build flags are defined :

- `ERRATA_A55_768277`: This applies errata 768277 workaround to Cortex-A55 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A55_778703`: This applies errata 778703 workaround to Cortex-A55 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A55_798797`: This applies errata 798797 workaround to Cortex-A55 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A55_846532`: This applies errata 846532 workaround to Cortex-A55 CPU. This needs to be enabled only for revision <= r0p1 of the CPU.
- `ERRATA_A55_903758`: This applies errata 903758 workaround to Cortex-A55 CPU. This needs to be enabled only for revision <= r0p1 of the CPU.
- `ERRATA_A55_1221012`: This applies errata 1221012 workaround to Cortex-A55 CPU. This needs to be enabled only for revision <= r1p0 of the CPU.
- `ERRATA_A55_1530923`: This applies errata 1530923 workaround to all revisions of Cortex-A55 CPU.

For Cortex-A57, the following errata build flags are defined :

- `ERRATA_A57_806969`: This applies errata 806969 workaround to Cortex-A57 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A57_813419`: This applies errata 813419 workaround to Cortex-A57 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A57_813420`: This applies errata 813420 workaround to Cortex-A57 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A57_814670`: This applies errata 814670 workaround to Cortex-A57 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A57_817169`: This applies errata 817169 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r0p1 of the CPU.
- `ERRATA_A57_826974`: This applies errata 826974 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r1p1 of the CPU.
- `ERRATA_A57_826977`: This applies errata 826977 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r1p1 of the CPU.
- `ERRATA_A57_828024`: This applies errata 828024 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r1p1 of the CPU.
- `ERRATA_A57_829520`: This applies errata 829520 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r1p2 of the CPU.
- `ERRATA_A57_833471`: This applies errata 833471 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r1p2 of the CPU.
- `ERRATA_A57_859972`: This applies errata 859972 workaround to Cortex-A57 CPU. This needs to be enabled only for revision <= r1p3 of the CPU.
- `ERRATA_A57_1319537`: This applies errata 1319537 workaround to all revisions of Cortex-A57 CPU.

For Cortex-A72, the following errata build flags are defined :

- `ERRATA_A72_859971`: This applies errata 859971 workaround to Cortex-A72 CPU. This needs to be enabled only for revision <= r0p3 of the CPU.
- `ERRATA_A72_1319367`: This applies errata 1319367 workaround to all revisions of Cortex-A72 CPU.

For Cortex-A73, the following errata build flags are defined :

- `ERRATA_A73_852427`: This applies errata 852427 workaround to Cortex-A73 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- `ERRATA_A73_855423`: This applies errata 855423 workaround to Cortex-A73 CPU. This needs to be enabled only for revision <= r0p1 of the CPU.

For Cortex-A75, the following errata build flags are defined :

- `ERRATA_A75_764081`: This applies errata 764081 workaround to Cortex-A75 CPU. This needs to be enabled only for revision r0p0 of the CPU.
- **`ERRATA_A75_790748`: This applies errata 790748 workaround to Cortex-A75** CPU. This needs to be enabled only for revision r0p0 of the CPU.

For Cortex-A76, the following errata build flags are defined :

- `ERRATA_A76_1073348`: This applies errata 1073348 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r1p0 of the CPU.
- `ERRATA_A76_1130799`: This applies errata 1130799 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r2p0 of the CPU.
- `ERRATA_A76_1220197`: This applies errata 1220197 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r2p0 of the CPU.
- `ERRATA_A76_1257314`: This applies errata 1257314 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- `ERRATA_A76_1262606`: This applies errata 1262606 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- `ERRATA_A76_1262888`: This applies errata 1262888 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- `ERRATA_A76_1275112`: This applies errata 1275112 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- `ERRATA_A76_1791580`: This applies errata 1791580 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r4p0 of the CPU.
- `ERRATA_A76_1800710`: This applies errata 1800710 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r4p0 of the CPU.
- `ERRATA_A76_1165522`: This applies errata 1165522 workaround to all revisions of Cortex-A76 CPU. This errata is fixed in r3p0 but due to limitation of errata framework this errata is applied to all revisions of Cortex-A76 CPU.
- `ERRATA_A76_1868343`: This applies errata 1868343 workaround to Cortex-A76 CPU. This needs to be enabled only for revision <= r4p0 of the CPU.

For Cortex-A77, the following errata build flags are defined :

- `ERRATA_A77_1508412`: This applies errata 1508412 workaround to Cortex-A77 CPU. This needs to be enabled only for revision <= r1p0 of the CPU.
- `ERRATA_A77_1800714`: This applies errata 1800714 workaround to Cortex-A77 CPU. This needs to be enabled only for revision <= r1p1 of the CPU.

- ERRATA_A77_1925769: This applies errata 1925769 workaround to Cortex-A77 CPU. This needs to be enabled only for revision <= r1p1 of the CPU.

For Cortex-A78, the following errata build flags are defined :

- ERRATA_A78_1688305: This applies errata 1688305 workaround to Cortex-A78 CPU. This needs to be enabled only for revision r0p0 - r1p0 of the CPU.

For Neoverse N1, the following errata build flags are defined :

- ERRATA_N1_1073348: This applies errata 1073348 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision r0p0 and r1p0 of the CPU.
- ERRATA_N1_1130799: This applies errata 1130799 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r2p0 of the CPU.
- ERRATA_N1_1165347: This applies errata 1165347 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r2p0 of the CPU.
- ERRATA_N1_1207823: This applies errata 1207823 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r2p0 of the CPU.
- ERRATA_N1_1220197: This applies errata 1220197 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r2p0 of the CPU.
- ERRATA_N1_1257314: This applies errata 1257314 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- ERRATA_N1_1262606: This applies errata 1262606 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- ERRATA_N1_1262888: This applies errata 1262888 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- ERRATA_N1_1275112: This applies errata 1275112 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- ERRATA_N1_1315703: This applies errata 1315703 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r3p0 of the CPU.
- ERRATA_N1_1542419: This applies errata 1542419 workaround to Neoverse-N1 CPU. This needs to be enabled only for revisions r3p0 - r4p0 of the CPU.
- ERRATA_N1_1868343: This applies errata 1868343 workaround to Neoverse-N1 CPU. This needs to be enabled only for revision <= r4p0 of the CPU.

5.3.3 DSU Errata Workarounds

Similar to CPU errata, TF-A also implements workarounds for DSU (DynamIQ Shared Unit) errata. The DSU errata details can be found in the respective Arm documentation:

- Arm DSU Software Developers Errata Notice.

Each erratum is identified by an ID, as defined in the DSU errata notice document. Thus, the build flags which enable/disable the errata workarounds have the format ERRATA_DSU_<ID>. The implementation and application logic of DSU errata workarounds are similar to *CPU errata workarounds*.

For DSU errata, the following build flags are defined:

- ERRATA_DSU_798953: This applies errata 798953 workaround for the affected DSU configurations. This errata applies only for those DSUs that revision is r0p0 (on r0p1 it is fixed). However, please note that this workaround results in increased DSU power consumption on idle.

- `ERRATA_DSU_936184`: This applies errata 936184 workaround for the affected DSU configurations. This errata applies only for those DSUs that contain the ACP interface **and** the DSU revision is older than r2p0 (on r2p0 it is fixed). However, please note that this workaround results in increased DSU power consumption on idle.

5.3.4 CPU Specific optimizations

This section describes some of the optimizations allowed by the CPU micro architecture that can be enabled by the platform as desired.

- `SKIP_A57_L1_FLUSH_PWR_DWN`: This flag enables an optimization in the Cortex-A57 cluster power down sequence by not flushing the Level 1 data cache. The L1 data cache and the L2 unified cache are inclusive. A flush of the L2 by set/way flushes any dirty lines from the L1 as well. This is a known safe deviation from the Cortex-A57 TRM defined power down sequence. Each Cortex-A57 based platform must make its own decision on whether to use the optimization.
- `A53_DISABLE_NON_TEMPORAL_HINT`: This flag disables the cache non-temporal hint. The LDNP/STNP instructions as implemented on Cortex-A53 do not behave in a way most programmers expect, and will most probably result in a significant speed degradation to any code that employs them. The Armv8-A architecture (see Arm DDI 0487A.h, section D3.4.3) allows cores to ignore the non-temporal hint and treat LDNP/STNP as LDP/STP instead. Enabling this flag enforces this behaviour. This needs to be enabled only for revisions <= r0p3 of the CPU and is enabled by default.
- `A57_DISABLE_NON_TEMPORAL_HINT`: This flag has the same behaviour as `A53_DISABLE_NON_TEMPORAL_HINT` but for Cortex-A57. This needs to be enabled only for revisions <= r1p2 of the CPU and is enabled by default, as recommended in section "4.7 Non-Temporal Loads/Stores" of the Cortex-A57 Software Optimization Guide.
- **''A57_ENABLE_NON_CACHEABLE_LOAD_FWD'': This flag enables non-cacheable** streaming enhancement feature for Cortex-A57 CPUs. Platforms can set this bit only if their memory system meets the requirement that cache line fill requests from the Cortex-A57 processor are atomic. Each Cortex-A57 based platform must make its own decision on whether to use the optimization. This flag is disabled by default.
- `NEOVERSE_N1_EXTERNAL_LLC`: This flag indicates that an external last level cache(LLC) is present in the system, and that the DataSource field on the master CHI interface indicates when data is returned from the LLC. This is used to control how the LL_CACHE* PMU events count.

5.4 Firmware Design

Trusted Firmware-A (TF-A) implements a subset of the Trusted Board Boot Requirements (TBBR) Platform Design Document (PDD) for Arm reference platforms.

The TBB sequence starts when the platform is powered on and runs up to the stage where it hands-off control to firmware running in the normal world in DRAM. This is the cold boot path.

TF-A also implements the Power State Coordination Interface PDD as a runtime service. PSCI is the interface from normal world software to firmware implementing power management use-cases (for example, secondary CPU boot, hotplug and idle). Normal world software can access TF-A runtime services via the Arm SMC (Secure Monitor Call) instruction. The SMC instruction must be used as mandated by the SMC Calling Convention (SMCCC).

TF-A implements a framework for configuring and managing interrupts generated in either security state. The details of the interrupt management framework and its design can be found in *Interrupt Management Framework*.

TF-A also implements a library for setting up and managing the translation tables. The details of this library can be found in *Translation (XLAT) Tables Library*.

TF-A can be built to support either AArch64 or AArch32 execution state.

5.4.1 Cold boot

The cold boot path starts when the platform is physically turned on. If `COLD_BOOT_SINGLE_CPU=0`, one of the CPUs released from reset is chosen as the primary CPU, and the remaining CPUs are considered secondary CPUs. The primary CPU is chosen through platform-specific means. The cold boot path is mainly executed by the primary CPU, other than essential CPU initialization executed by all CPUs. The secondary CPUs are kept in a safe platform-specific state until the primary CPU has performed enough initialization to boot them.

Refer to the *CPU Reset* for more information on the effect of the `COLD_BOOT_SINGLE_CPU` platform build option.

The cold boot path in this implementation of TF-A depends on the execution state. For AArch64, it is divided into five steps (in order of execution):

- Boot Loader stage 1 (BL1) *AP Trusted ROM*
- Boot Loader stage 2 (BL2) *Trusted Boot Firmware*
- Boot Loader stage 3-1 (BL31) *EL3 Runtime Software*
- Boot Loader stage 3-2 (BL32) *Secure-EL1 Payload* (optional)
- Boot Loader stage 3-3 (BL33) *Non-trusted Firmware*

For AArch32, it is divided into four steps (in order of execution):

- Boot Loader stage 1 (BL1) *AP Trusted ROM*
- Boot Loader stage 2 (BL2) *Trusted Boot Firmware*
- Boot Loader stage 3-2 (BL32) *EL3 Runtime Software*
- Boot Loader stage 3-3 (BL33) *Non-trusted Firmware*

Arm development platforms (Fixed Virtual Platforms (FVPs) and Juno) implement a combination of the following types of memory regions. Each bootloader stage uses one or more of these memory regions.

- Regions accessible from both non-secure and secure states. For example, non-trusted SRAM, ROM and DRAM.
- Regions accessible from only the secure state. For example, trusted SRAM and ROM. The FVPs also implement the trusted DRAM which is statically configured. Additionally, the Base FVPs and Juno development platform configure the TrustZone Controller (TZC) to create a region in the DRAM which is accessible only from the secure state.

The sections below provide the following details:

- dynamic configuration of Boot Loader stages
- initialization and execution of the first three stages during cold boot
- specification of the EL3 Runtime Software (BL31 for AArch64 and BL32 for AArch32) entrypoint requirements for use by alternative Trusted Boot Firmware in place of the provided BL1 and BL2

Dynamic Configuration during cold boot

Each of the Boot Loader stages may be dynamically configured if required by the platform. The Boot Loader stage may optionally specify a firmware configuration file and/or hardware configuration file as listed below:

- FW_CONFIG - The firmware configuration file. Holds properties shared across all BLx images. An example is the "dtb-registry" node, which contains the information about the other device tree configurations (load-address, size, image_id).
- HW_CONFIG - The hardware configuration file. Can be shared by all Boot Loader stages and also by the Normal World Rich OS.
- TB_FW_CONFIG - Trusted Boot Firmware configuration file. Shared between BL1 and BL2.
- SOC_FW_CONFIG - SoC Firmware configuration file. Used by BL31.
- TOS_FW_CONFIG - Trusted OS Firmware configuration file. Used by Trusted OS (BL32).
- NT_FW_CONFIG - Non Trusted Firmware configuration file. Used by Non-trusted firmware (BL33).

The Arm development platforms use the Flattened Device Tree format for the dynamic configuration files.

Each Boot Loader stage can pass up to 4 arguments via registers to the next stage. BL2 passes the list of the next images to execute to the *EL3 Runtime Software* (BL31 for AArch64 and BL32 for AArch32) via *arg0*. All the other arguments are platform defined. The Arm development platforms use the following convention:

- BL1 passes the address of a meminfo_t structure to BL2 via `arg1`. This structure contains the memory layout available to BL2.
- When dynamic configuration files are present, the firmware configuration for the next Boot Loader stage is populated in the first available argument and the generic hardware configuration is passed the next available argument. For example,
  - FW_CONFIG is loaded by BL1, then its address is passed in `arg0` to BL2.
  - TB_FW_CONFIG address is retrieved by BL2 from FW_CONFIG device tree.
  - If HW_CONFIG is loaded by BL1, then its address is passed in `arg2` to BL2. Note, `arg1` is already used for meminfo_t.
  - If SOC_FW_CONFIG is loaded by BL2, then its address is passed in `arg1` to BL31. Note, `arg0` is used to pass the list of executable images.
  - Similarly, if HW_CONFIG is loaded by BL1 or BL2, then its address is passed in `arg2` to BL31.
  - For other BL3x images, if the firmware configuration file is loaded by BL2, then its address is passed in `arg0` and if HW_CONFIG is loaded then its address is passed in `arg1`.

BL1

This stage begins execution from the platform's reset vector at EL3. The reset address is platform dependent but it is usually located in a Trusted ROM area. The BL1 data section is copied to trusted SRAM at runtime.

On the Arm development platforms, BL1 code starts execution from the reset vector defined by the constant `BL1_RO_BASE`. The BL1 data section is copied to the top of trusted SRAM as defined by the constant `BL1_RW_BASE`.

The functionality implemented by this stage is as follows.

Determination of boot path

Whenever a CPU is released from reset, BL1 needs to distinguish between a warm boot and a cold boot. This is done using platform-specific mechanisms (see the `plat_get_my_entrypoint()` function in the *Porting Guide*). In the case of a warm boot, a CPU is expected to continue execution from a separate entrypoint. In the case of a cold boot, the secondary CPUs are placed in a safe platform-specific state (see the `plat_secondary_cold_boot_setup()` function in the *Porting Guide*) while the primary CPU executes the remaining cold boot path as described in the following sections.

This step only applies when `PROGRAMMABLE_RESET_ADDRESS=0`. Refer to the *CPU Reset* for more information on the effect of the `PROGRAMMABLE_RESET_ADDRESS` platform build option.

Architectural initialization

BL1 performs minimal architectural initialization as follows.

- Exception vectors

  BL1 sets up simple exception vectors for both synchronous and asynchronous exceptions. The default behavior upon receiving an exception is to populate a status code in the general purpose register `X0/R0` and call the `plat_report_exception()` function (see the *Porting Guide*). The status code is one of:

  For AArch64:

```
0x0 : Synchronous exception from Current EL with SP_EL0
0x1 : IRQ exception from Current EL with SP_EL0
0x2 : FIQ exception from Current EL with SP_EL0
0x3 : System Error exception from Current EL with SP_EL0
0x4 : Synchronous exception from Current EL with SP_ELx
0x5 : IRQ exception from Current EL with SP_ELx
0x6 : FIQ exception from Current EL with SP_ELx
0x7 : System Error exception from Current EL with SP_ELx
0x8 : Synchronous exception from Lower EL using aarch64
0x9 : IRQ exception from Lower EL using aarch64
0xa : FIQ exception from Lower EL using aarch64
0xb : System Error exception from Lower EL using aarch64
0xc : Synchronous exception from Lower EL using aarch32
0xd : IRQ exception from Lower EL using aarch32
0xe : FIQ exception from Lower EL using aarch32
0xf : System Error exception from Lower EL using aarch32
```

  For AArch32:

```
0x10 : User mode
0x11 : FIQ mode
0x12 : IRQ mode
0x13 : SVC mode
0x16 : Monitor mode
0x17 : Abort mode
0x1a : Hypervisor mode
0x1b : Undefined mode
0x1f : System mode
```

  The `plat_report_exception()` implementation on the Arm FVP port programs the Versatile Express System LED register in the following format to indicate the occurrence of an unexpected exception:

```
SYS_LED[0]   - Security state (Secure=0/Non-Secure=1)
SYS_LED[2:1] - Exception Level (EL3=0x3, EL2=0x2, EL1=0x1, EL0=0x0)
               For AArch32 it is always 0x0
SYS_LED[7:3] - Exception Class (Sync/Async & origin). This is the value
               of the status code
```

A write to the LED register reflects in the System LEDs (S6LED0..7) in the CLCD window of the FVP.

BL1 does not expect to receive any exceptions other than the SMC exception. For the latter, BL1 installs a simple stub. The stub expects to receive a limited set of SMC types (determined by their function IDs in the general purpose register `X0/R0`):

- `BL1_SMC_RUN_IMAGE`: This SMC is raised by BL2 to make BL1 pass control to EL3 Runtime Software.
- All SMCs listed in section "BL1 SMC Interface" in the *Firmware Update (FWU)* Design Guide are supported for AArch64 only. These SMCs are currently not supported when BL1 is built for AArch32.

Any other SMC leads to an assertion failure.

- CPU initialization

  BL1 calls the `reset_handler()` function which in turn calls the CPU specific reset handler function (see the section: "CPU specific operations framework").

- Control register setup (for AArch64)
  - `SCTLR_EL3`. Instruction cache is enabled by setting the `SCTLR_EL3.I` bit. Alignment and stack alignment checking is enabled by setting the `SCTLR_EL3.A` and `SCTLR_EL3.SA` bits. Exception endianness is set to little-endian by clearing the `SCTLR_EL3.EE` bit.
  - `SCR_EL3`. The register width of the next lower exception level is set to AArch64 by setting the `SCR.RW` bit. The `SCR.EA` bit is set to trap both External Aborts and SError Interrupts in EL3. The `SCR.SIF` bit is also set to disable instruction fetches from Non-secure memory when in secure state.
  - `CPTR_EL3`. Accesses to the `CPACR_EL1` register from EL1 or EL2, or the `CPTR_EL2` register from EL2 are configured to not trap to EL3 by clearing the `CPTR_EL3.TCPAC` bit. Access to the trace functionality is configured not to trap to EL3 by clearing the `CPTR_EL3.TTA` bit. Instructions that access the registers associated with Floating Point and Advanced SIMD execution are configured to not trap to EL3 by clearing the `CPTR_EL3.TFP` bit.
  - `DAIF`. The SError interrupt is enabled by clearing the SError interrupt mask bit.
  - `MDCR_EL3`. The trap controls, `MDCR_EL3.TDOSA`, `MDCR_EL3.TDA` and `MDCR_EL3.TPM`, are set so that accesses to the registers they control do not trap to EL3. AArch64 Secure self-hosted debug is disabled by setting the `MDCR_EL3.SDD` bit. Also `MDCR_EL3.SPD32` is set to disable AArch32 Secure self-hosted privileged debug from S-EL1.
- Control register setup (for AArch32)
  - `SCTLR`. Instruction cache is enabled by setting the `SCTLR.I` bit. Alignment checking is enabled by setting the `SCTLR.A` bit. Exception endianness is set to little-endian by clearing the `SCTLR.EE` bit.
  - `SCR`. The `SCR.SIF` bit is set to disable instruction fetches from Non-secure memory when in secure state.
  - `CPACR`. Allow execution of Advanced SIMD instructions at PL0 and PL1, by clearing the `CPACR.ASEDIS` bit. Access to the trace functionality is configured not to trap to undefined mode by clearing the `CPACR.TRCDIS` bit.
  - `NSACR`. Enable non-secure access to Advanced SIMD functionality and system register access to implemented trace registers.

  - `FPEXC`. Enable access to the Advanced SIMD and floating-point functionality from all Exception levels.
  - `CPSR.A`. The Asynchronous data abort interrupt is enabled by clearing the Asynchronous data abort interrupt mask bit.
  - `SDCR`. The `SDCR.SPD` field is set to disable AArch32 Secure self-hosted privileged debug.

Platform initialization

On Arm platforms, BL1 performs the following platform initializations:

- Enable the Trusted Watchdog.
- Initialize the console.
- Configure the Interconnect to enable hardware coherency.
- Enable the MMU and map the memory it needs to access.
- Configure any required platform storage to load the next bootloader image (BL2).
- If the BL1 dynamic configuration file, `TB_FW_CONFIG`, is available, then load it to the platform defined address and make it available to BL2 via `arg0`.
- Configure the system timer and program the *CNTFRQ_EL0* for use by NS-BL1U and NS-BL2U firmware update images.

Firmware Update detection and execution

After performing platform setup, BL1 common code calls `bl1_plat_get_next_image_id()` to determine if *Firmware Update (FWU)* is required or to proceed with the normal boot process. If the platform code returns `BL2_IMAGE_ID` then the normal boot sequence is executed as described in the next section, else BL1 assumes that *Firmware Update (FWU)* is required and execution passes to the first image in the *Firmware Update (FWU)* process. In either case, BL1 retrieves a descriptor of the next image by calling `bl1_plat_get_image_desc()`. The image descriptor contains an `entry_point_info_t` structure, which BL1 uses to initialize the execution state of the next image.

BL2 image load and execution

In the normal boot flow, BL1 execution continues as follows:

1. BL1 prints the following string from the primary CPU to indicate successful execution of the BL1 stage:

   ```
   "Booting Trusted Firmware"
   ```

2. BL1 loads a BL2 raw binary image from platform storage, at a platform-specific base address. Prior to the load, BL1 invokes `bl1_plat_handle_pre_image_load()` which allows the platform to update or use the image information. If the BL2 image file is not present or if there is not enough free trusted SRAM the following error message is printed:

   ```
   "Failed to load BL2 firmware."
   ```

3. BL1 invokes `bl1_plat_handle_post_image_load()` which again is intended for platforms to take further action after image load. This function must populate the necessary arguments for BL2, which may also include the memory layout. Further description of the memory layout can be found later in this document.

4. BL1 passes control to the BL2 image at Secure EL1 (for AArch64) or at Secure SVC mode (for AArch32), starting from its load address.

BL2

BL1 loads and passes control to BL2 at Secure-EL1 (for AArch64) or at Secure SVC mode (for AArch32) . BL2 is linked against and loaded at a platform-specific base address (more information can be found later in this document). The functionality implemented by BL2 is as follows.

Architectural initialization

For AArch64, BL2 performs the minimal architectural initialization required for subsequent stages of TF-A and normal world software. EL1 and EL0 are given access to Floating Point and Advanced SIMD registers by setting the `CPACR.FPEN` bits.

For AArch32, the minimal architectural initialization required for subsequent stages of TF-A and normal world software is taken care of in BL1 as both BL1 and BL2 execute at PL1.

Platform initialization

On Arm platforms, BL2 performs the following platform initializations:

- Initialize the console.
- Configure any required platform storage to allow loading further bootloader images.
- Enable the MMU and map the memory it needs to access.
- Perform platform security setup to allow access to controlled components.
- Reserve some memory for passing information to the next bootloader image EL3 Runtime Software and populate it.
- Define the extents of memory available for loading each subsequent bootloader image.
- If BL1 has passed TB_FW_CONFIG dynamic configuration file in `arg0`, then parse it.

Image loading in BL2

BL2 generic code loads the images based on the list of loadable images provided by the platform. BL2 passes the list of executable images provided by the platform to the next handover BL image.

The list of loadable images provided by the platform may also contain dynamic configuration files. The files are loaded and can be parsed as needed in the `bl2_plat_handle_post_image_load()` function. These configuration files can be passed to next Boot Loader stages as arguments by updating the corresponding entrypoint information in this function.

SCP_BL2 (System Control Processor Firmware) image load

Some systems have a separate System Control Processor (SCP) for power, clock, reset and system control. BL2 loads the optional SCP_BL2 image from platform storage into a platform-specific region of secure memory. The subsequent handling of SCP_BL2 is platform specific. For example, on the Juno Arm development platform port the image is transferred into SCP's internal memory using the Boot Over MHU (BOM) protocol after being loaded in the trusted SRAM memory. The SCP executes SCP_BL2 and signals to the Application Processor (AP) for BL2 execution to continue.

EL3 Runtime Software image load

BL2 loads the EL3 Runtime Software image from platform storage into a platform- specific address in trusted SRAM. If there is not enough memory to load the image or image is missing it leads to an assertion failure.

AArch64 BL32 (Secure-EL1 Payload) image load

BL2 loads the optional BL32 image from platform storage into a platform- specific region of secure memory. The image executes in the secure world. BL2 relies on BL31 to pass control to the BL32 image, if present. Hence, BL2 populates a platform-specific area of memory with the entrypoint/load-address of the BL32 image. The value of the Saved Processor Status Register (`SPSR`) for entry into BL32 is not determined by BL2, it is initialized by the Secure-EL1 Payload Dispatcher (see later) within BL31, which is responsible for managing interaction with BL32. This information is passed to BL31.

BL33 (Non-trusted Firmware) image load

BL2 loads the BL33 image (e.g. UEFI or other test or boot software) from platform storage into non-secure memory as defined by the platform.

BL2 relies on EL3 Runtime Software to pass control to BL33 once secure state initialization is complete. Hence, BL2 populates a platform-specific area of memory with the entrypoint and Saved Program Status Register (`SPSR`) of the normal world software image. The entrypoint is the load address of the BL33 image. The `SPSR` is determined as specified in Section 5.13 of the Power State Coordination Interface PDD. This information is passed to the EL3 Runtime Software.

AArch64 BL31 (EL3 Runtime Software) execution

BL2 execution continues as follows:

1. BL2 passes control back to BL1 by raising an SMC, providing BL1 with the BL31 entrypoint. The exception is handled by the SMC exception handler installed by BL1.
2. BL1 turns off the MMU and flushes the caches. It clears the `SCTLR_EL3.M/I/C` bits, flushes the data cache to the point of coherency and invalidates the TLBs.
3. BL1 passes control to BL31 at the specified entrypoint at EL3.

Running BL2 at EL3 execution level

Some platforms have a non-TF-A Boot ROM that expects the next boot stage to execute at EL3. On these platforms, TF-A BL1 is a waste of memory as its only purpose is to ensure TF-A BL2 is entered at S-EL1. To avoid this waste, a special mode enables BL2 to execute at EL3, which allows a non-TF-A Boot ROM to load and jump directly to BL2. This mode is selected when the build flag BL2_AT_EL3 is enabled. The main differences in this mode are:

1. BL2 includes the reset code and the mailbox mechanism to differentiate cold boot and warm boot. It runs at EL3 doing the arch initialization required for EL3.
2. BL2 does not receive the meminfo information from BL1 anymore. This information can be passed by the Boot ROM or be internal to the BL2 image.
3. Since BL2 executes at EL3, BL2 jumps directly to the next image, instead of invoking the RUN_IMAGE SMC call.

We assume 3 different types of BootROM support on the platform:

1. The Boot ROM always jumps to the same address, for both cold and warm boot. In this case, we will need to keep a resident part of BL2 whose memory cannot be reclaimed by any other image. The linker script defines the symbols __TEXT_RESIDENT_START__ and __TEXT_RESIDENT_END__ that allows the platform to configure correctly the memory map.
2. The platform has some mechanism to indicate the jump address to the Boot ROM. Platform code can then program the jump address with psci_warmboot_entrypoint during cold boot.
3. The platform has some mechanism to program the reset address using the PROGRAMMABLE_RESET_ADDRESS feature. Platform code can then program the reset address with psci_warmboot_entrypoint during cold boot, bypassing the boot ROM for warm boot.

In the last 2 cases, no part of BL2 needs to remain resident at runtime. In the first 2 cases, we expect the Boot ROM to be able to differentiate between warm and cold boot, to avoid loading BL2 again during warm boot.

This functionality can be tested with FVP loading the image directly in memory and changing the address where the system jumps at reset. For example:

> -C cluster0.cpu0.RVBAR=0x4022000 –data cluster0.cpu0=bl2.bin@0x4022000

With this configuration, FVP is like a platform of the first case, where the Boot ROM jumps always to the same address. For simplification, BL32 is loaded in DRAM in this case, to avoid other images reclaiming BL2 memory.

AArch64 BL31

The image for this stage is loaded by BL2 and BL1 passes control to BL31 at EL3. BL31 executes solely in trusted SRAM. BL31 is linked against and loaded at a platform-specific base address (more information can be found later in this document). The functionality implemented by BL31 is as follows.

Architectural initialization

Currently, BL31 performs a similar architectural initialization to BL1 as far as system register settings are concerned. Since BL1 code resides in ROM, architectural initialization in BL31 allows override of any previous initialization done by BL1.

BL31 initializes the per-CPU data framework, which provides a cache of frequently accessed per-CPU data optimised for fast, concurrent manipulation on different CPUs. This buffer includes pointers to per-CPU contexts, crash buffer, CPU reset and power down operations, PSCI data, platform data and so on.

It then replaces the exception vectors populated by BL1 with its own. BL31 exception vectors implement more elaborate support for handling SMCs since this is the only mechanism to access the runtime services implemented by

BL31 (PSCI for example). BL31 checks each SMC for validity as specified by the SMC Calling Convention before passing control to the required SMC handler routine.

BL31 programs the `CNTFRQ_EL0` register with the clock frequency of the system counter, which is provided by the platform.

Platform initialization

BL31 performs detailed platform initialization, which enables normal world software to function correctly.

On Arm platforms, this consists of the following:

- Initialize the console.
- Configure the Interconnect to enable hardware coherency.
- Enable the MMU and map the memory it needs to access.
- Initialize the generic interrupt controller.
- Initialize the power controller device.
- Detect the system topology.

Runtime services initialization

BL31 is responsible for initializing the runtime services. One of them is PSCI.

As part of the PSCI initializations, BL31 detects the system topology. It also initializes the data structures that implement the state machine used to track the state of power domain nodes. The state can be one of `OFF`, `RUN` or `RETENTION`. All secondary CPUs are initially in the `OFF` state. The cluster that the primary CPU belongs to is `ON`; any other cluster is `OFF`. It also initializes the locks that protect them. BL31 accesses the state of a CPU or cluster immediately after reset and before the data cache is enabled in the warm boot path. It is not currently possible to use 'exclusive' based spinlocks, therefore BL31 uses locks based on Lamport's Bakery algorithm instead.

The runtime service framework and its initialization is described in more detail in the "EL3 runtime services framework" section below.

Details about the status of the PSCI implementation are provided in the "Power State Coordination Interface" section below.

AArch64 BL32 (Secure-EL1 Payload) image initialization

If a BL32 image is present then there must be a matching Secure-EL1 Payload Dispatcher (SPD) service (see later for details). During initialization that service must register a function to carry out initialization of BL32 once the runtime services are fully initialized. BL31 invokes such a registered function to initialize BL32 before running BL33. This initialization is not necessary for AArch32 SPs.

Details on BL32 initialization and the SPD's role are described in the *Secure-EL1 Payloads and Dispatchers* section below.

BL33 (Non-trusted Firmware) execution

EL3 Runtime Software initializes the EL2 or EL1 processor context for normal- world cold boot, ensuring that no secure state information finds its way into the non-secure execution state. EL3 Runtime Software uses the entrypoint information provided by BL2 to jump to the Non-trusted firmware image (BL33) at the highest available Exception Level (EL2 if available, otherwise EL1).

Using alternative Trusted Boot Firmware in place of BL1 & BL2 (AArch64 only)

Some platforms have existing implementations of Trusted Boot Firmware that would like to use TF-A BL31 for the EL3 Runtime Software. To enable this firmware architecture it is important to provide a fully documented and stable interface between the Trusted Boot Firmware and BL31.

Future changes to the BL31 interface will be done in a backwards compatible way, and this enables these firmware components to be independently enhanced/ updated to develop and exploit new functionality.

Required CPU state when calling `bl31_entrypoint()` during cold boot

This function must only be called by the primary CPU.

On entry to this function the calling primary CPU must be executing in AArch64 EL3, little-endian data access, and all interrupt sources masked:

```
PSTATE.EL = 3
PSTATE.RW = 1
PSTATE.DAIF = 0xf
SCTLR_EL3.EE = 0
```

X0 and X1 can be used to pass information from the Trusted Boot Firmware to the platform code in BL31:

```
X0 : Reserved for common TF-A information
X1 : Platform specific information
```

BL31 zero-init sections (e.g. `.bss`) should not contain valid data on entry, these will be zero filled prior to invoking platform setup code.

Use of the X0 and X1 parameters

The parameters are platform specific and passed from `bl31_entrypoint()` to `bl31_early_platform_setup()`. The value of these parameters is never directly used by the common BL31 code.

The convention is that `X0` conveys information regarding the BL31, BL32 and BL33 images from the Trusted Boot firmware and `X1` can be used for other platform specific purpose. This convention allows platforms which use TF-A's BL1 and BL2 images to transfer additional platform specific information from Secure Boot without conflicting with future evolution of TF-A using `X0` to pass a `bl31_params` structure.

BL31 common and SPD initialization code depends on image and entrypoint information about BL33 and BL32, which is provided via BL31 platform APIs. This information is required until the start of execution of BL33. This information can be provided in a platform defined manner, e.g. compiled into the platform code in BL31, or provided in a platform defined memory location by the Trusted Boot firmware, or passed from the Trusted Boot Firmware via the Cold boot Initialization parameters. This data may need to be cleaned out of the CPU caches if it is provided by an earlier boot stage and then accessed by BL31 platform code before the caches are enabled.

TF-A's BL2 implementation passes a `bl31_params` structure in `X0` and the Arm development platforms interpret this in the BL31 platform code.

MMU, Data caches & Coherency

BL31 does not depend on the enabled state of the MMU, data caches or interconnect coherency on entry to `bl31_entrypoint()`. If these are disabled on entry, these should be enabled during `bl31_plat_arch_setup()`.

Data structures used in the BL31 cold boot interface

These structures are designed to support compatibility and independent evolution of the structures and the firmware images. For example, a version of BL31 that can interpret the BL3x image information from different versions of BL2, a platform that uses an extended entry_point_info structure to convey additional register information to BL31, or a ELF image loader that can convey more details about the firmware images.

To support these scenarios the structures are versioned and sized, which enables BL31 to detect which information is present and respond appropriately. The `param_header` is defined to capture this information:

```
typedef struct param_header {
    uint8_t type;       /* type of the structure */
    uint8_t version;    /* version of this structure */
    uint16_t size;      /* size of this structure in bytes */
    uint32_t attr;      /* attributes: unused bits SBZ */
} param_header_t;
```

The structures using this format are `entry_point_info`, `image_info` and `bl31_params`. The code that allocates and populates these structures must set the header fields appropriately, and the `SET_PARAM_HEAD()` a macro is defined to simplify this action.

Required CPU state for BL31 Warm boot initialization

When requesting a CPU power-on, or suspending a running CPU, TF-A provides the platform power management code with a Warm boot initialization entry-point, to be invoked by the CPU immediately after the reset handler. On entry to the Warm boot initialization function the calling CPU must be in AArch64 EL3, little-endian data access and all interrupt sources masked:

```
PSTATE.EL = 3
PSTATE.RW = 1
PSTATE.DAIF = 0xf
SCTLR_EL3.EE = 0
```

The PSCI implementation will initialize the processor state and ensure that the platform power management code is then invoked as required to initialize all necessary system, cluster and CPU resources.

AArch32 EL3 Runtime Software entrypoint interface

To enable this firmware architecture it is important to provide a fully documented and stable interface between the Trusted Boot Firmware and the AArch32 EL3 Runtime Software.

Future changes to the entrypoint interface will be done in a backwards compatible way, and this enables these firmware components to be independently enhanced/updated to develop and exploit new functionality.

Required CPU state when entering during cold boot

This function must only be called by the primary CPU.

On entry to this function the calling primary CPU must be executing in AArch32 EL3, little-endian data access, and all interrupt sources masked:

```
PSTATE.AIF = 0x7
SCTLR.EE = 0
```

R0 and R1 are used to pass information from the Trusted Boot Firmware to the platform code in AArch32 EL3 Runtime Software:

```
R0 : Reserved for common TF-A information
R1 : Platform specific information
```

Use of the R0 and R1 parameters

The parameters are platform specific and the convention is that `R0` conveys information regarding the BL3x images from the Trusted Boot firmware and `R1` can be used for other platform specific purpose. This convention allows platforms which use TF-A's BL1 and BL2 images to transfer additional platform specific information from Secure Boot without conflicting with future evolution of TF-A using `R0` to pass a `bl_params` structure.

The AArch32 EL3 Runtime Software is responsible for entry into BL33. This information can be obtained in a platform defined manner, e.g. compiled into the AArch32 EL3 Runtime Software, or provided in a platform defined memory location by the Trusted Boot firmware, or passed from the Trusted Boot Firmware via the Cold boot Initialization parameters. This data may need to be cleaned out of the CPU caches if it is provided by an earlier boot stage and then accessed by AArch32 EL3 Runtime Software before the caches are enabled.

When using AArch32 EL3 Runtime Software, the Arm development platforms pass a `bl_params` structure in `R0` from BL2 to be interpreted by AArch32 EL3 Runtime Software platform code.

MMU, Data caches & Coherency

AArch32 EL3 Runtime Software must not depend on the enabled state of the MMU, data caches or interconnect coherency in its entrypoint. They must be explicitly enabled if required.

Data structures used in cold boot interface

The AArch32 EL3 Runtime Software cold boot interface uses `bl_params` instead of `bl31_params`. The `bl_params` structure is based on the convention described in AArch64 BL31 cold boot interface section.

Required CPU state for warm boot initialization

When requesting a CPU power-on, or suspending a running CPU, AArch32 EL3 Runtime Software must ensure execution of a warm boot initialization entrypoint. If TF-A BL1 is used and the PROGRAMMABLE_RESET_ADDRESS build flag is false, then AArch32 EL3 Runtime Software must ensure that BL1 branches to the warm boot entrypoint by arranging for the BL1 platform function, plat_get_my_entrypoint(), to return a non-zero value.

In this case, the warm boot entrypoint must be in AArch32 EL3, little-endian data access and all interrupt sources masked:

```
PSTATE.AIF = 0x7
SCTLR.EE = 0
```

The warm boot entrypoint may be implemented by using TF-A `psci_warmboot_entrypoint()` function. In that case, the platform must fulfil the pre-requisites mentioned in the *PSCI Library Integration guide for Armv8-A AArch32 systems*.

5.4.2 EL3 runtime services framework

Software executing in the non-secure state and in the secure state at exception levels lower than EL3 will request runtime services using the Secure Monitor Call (SMC) instruction. These requests will follow the convention described in the SMC Calling Convention PDD (SMCCC). The SMCCC assigns function identifiers to each SMC request and describes how arguments are passed and returned.

The EL3 runtime services framework enables the development of services by different providers that can be easily integrated into final product firmware. The following sections describe the framework which facilitates the registration, initialization and use of runtime services in EL3 Runtime Software (BL31).

The design of the runtime services depends heavily on the concepts and definitions described in the SMCCC, in particular SMC Function IDs, Owning Entity Numbers (OEN), Fast and Yielding calls, and the SMC32 and SMC64 calling conventions. Please refer to that document for more detailed explanation of these terms.

The following runtime services are expected to be implemented first. They have not all been instantiated in the current implementation.

1. Standard service calls

   This service is for management of the entire system. The Power State Coordination Interface (PSCI) is the first set of standard service calls defined by Arm (see PSCI section later).

2. Secure-EL1 Payload Dispatcher service

   If a system runs a Trusted OS or other Secure-EL1 Payload (SP) then it also requires a *Secure Monitor* at EL3 to switch the EL1 processor context between the normal world (EL1/EL2) and trusted world (Secure-EL1). The Secure Monitor will make these world switches in response to SMCs. The SMCCC provides for such SMCs with the Trusted OS Call and Trusted Application Call OEN ranges.

   The interface between the EL3 Runtime Software and the Secure-EL1 Payload is not defined by the SMCCC or any other standard. As a result, each Secure-EL1 Payload requires a specific Secure Monitor that runs as a runtime service - within TF-A this service is referred to as the Secure-EL1 Payload Dispatcher (SPD).

TF-A provides a Test Secure-EL1 Payload (TSP) and its associated Dispatcher (TSPD). Details of SPD design and TSP/TSPD operation are described in the *Secure-EL1 Payloads and Dispatchers* section below.

3. CPU implementation service

   This service will provide an interface to CPU implementation specific services for a given platform e.g. access to processor errata workarounds. This service is currently unimplemented.

Additional services for Arm Architecture, SiP and OEM calls can be implemented. Each implemented service handles a range of SMC function identifiers as described in the SMCCC.

Registration

A runtime service is registered using the `DECLARE_RT_SVC()` macro, specifying the name of the service, the range of OENs covered, the type of service and initialization and call handler functions. This macro instantiates a `const struct rt_svc_desc` for the service with these details (see `runtime_svc.h`). This structure is allocated in a special ELF section `rt_svc_descs`, enabling the framework to find all service descriptors included into BL31.

The specific service for a SMC Function is selected based on the OEN and call type of the Function ID, and the framework uses that information in the service descriptor to identify the handler for the SMC Call.

The service descriptors do not include information to identify the precise set of SMC function identifiers supported by this service implementation, the security state from which such calls are valid nor the capability to support 64-bit and/or 32-bit callers (using SMC32 or SMC64). Responding appropriately to these aspects of a SMC call is the responsibility of the service implementation, the framework is focused on integration of services from different providers and minimizing the time taken by the framework before the service handler is invoked.

Details of the parameters, requirements and behavior of the initialization and call handling functions are provided in the following sections.

Initialization

`runtime_svc_init()` in `runtime_svc.c` initializes the runtime services framework running on the primary CPU during cold boot as part of the BL31 initialization. This happens prior to initializing a Trusted OS and running Normal world boot firmware that might in turn use these services. Initialization involves validating each of the declared runtime service descriptors, calling the service initialization function and populating the index used for runtime lookup of the service.

The BL31 linker script collects all of the declared service descriptors into a single array and defines symbols that allow the framework to locate and traverse the array, and determine its size.

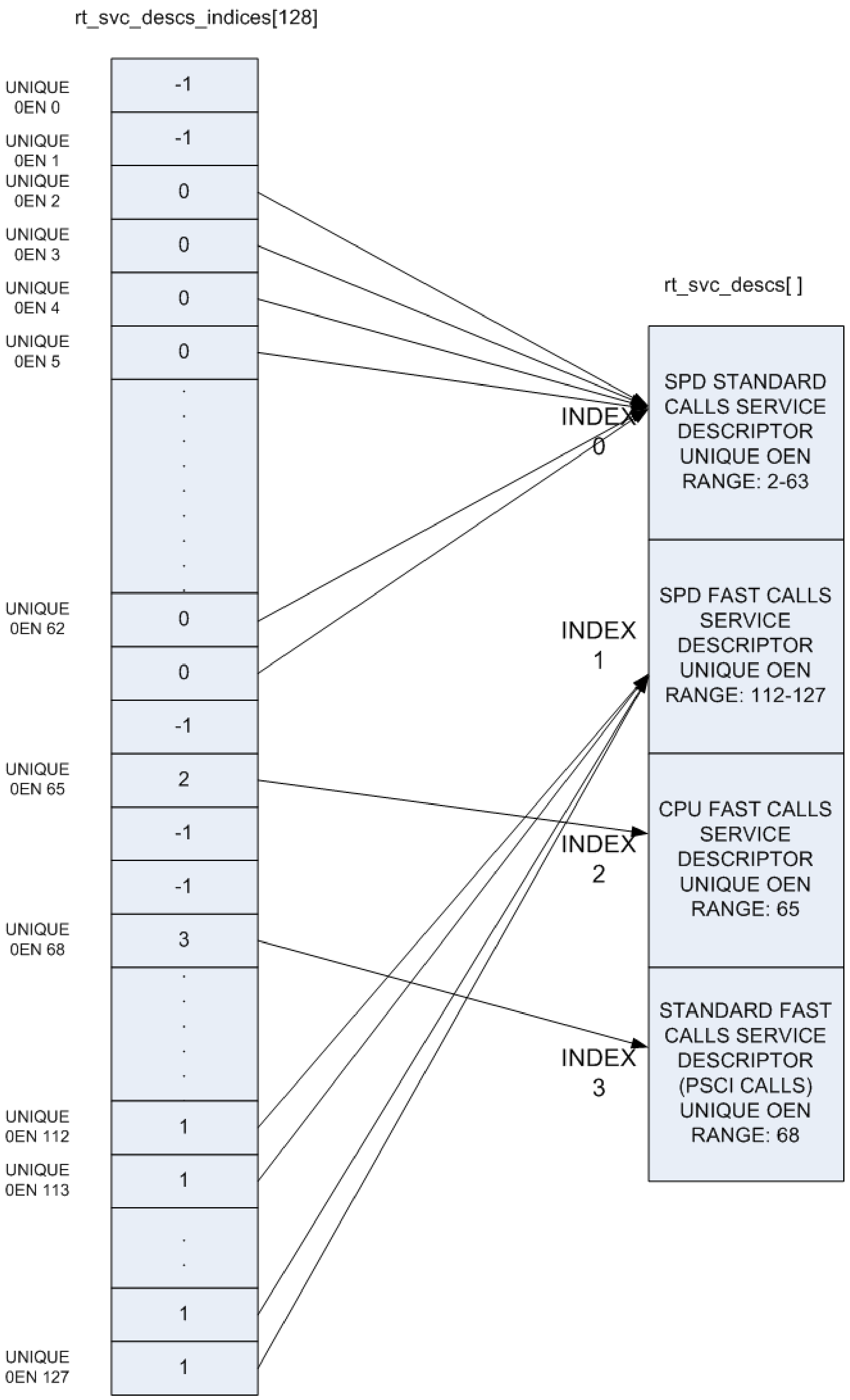
The framework does basic validation of each descriptor to halt firmware initialization if service declaration errors are detected. The framework does not check descriptors for the following error conditions, and may behave in an unpredictable manner under such scenarios:

1. Overlapping OEN ranges
2. Multiple descriptors for the same range of OENs and `call_type`
3. Incorrect range of owning entity numbers for a given `call_type`

Once validated, the service `init()` callback is invoked. This function carries out any essential EL3 initialization before servicing requests. The `init()` function is only invoked on the primary CPU during cold boot. If the service uses per-CPU data this must either be initialized for all CPUs during this call, or be done lazily when a CPU first issues an SMC call to that service. If `init()` returns anything other than `0`, this is treated as an initialization error and the service is ignored: this does not cause the firmware to halt.

The OEN and call type fields present in the SMC Function ID cover a total of 128 distinct services, but in practice a single descriptor can cover a range of OENs, e.g. SMCs to call a Trusted OS function. To optimize the lookup of a service handler, the framework uses an array of 128 indices that map every distinct OEN/call-type combination either to one of the declared services or to indicate the service is not handled. This `rt_svc_descs_indices[]` array is populated for all of the OENs covered by a service after the service `init()` function has reported success. So a service that fails to initialize will never have it's `handle()` function invoked.

The following figure shows how the `rt_svc_descs_indices[]` index maps the SMC Function ID call type and OEN onto a specific service handler in the `rt_svc_descs[]` array.


rt_svc_descs_indices[128]
UNIQUE 0EN 0
UNIQUE 0EN 1
UNIQUE 0EN 2
UNIQUE 0EN 3
UNIQUE 0EN 4
UNIQUE 0EN 5
UNIQUE 0EN 62
UNIQUE 0EN 65
UNIQUE 0EN 68
UNIQUE 0EN 112
UNIQUE 0EN 113
UNIQUE 0EN 127
-1
-1
0
0
0
0
0
0
-1
2
-1
-1
3
1
1
1
1
rt_svc_descs[ ]
INDEX 0
INDEX 1
INDEX 2
INDEX 3
SPD STANDARD CALLS SERVICE DESCRIPTOR UNIQUE OEN RANGE: 2-63
SPD FAST CALLS SERVICE DESCRIPTOR UNIQUE OEN RANGE: 112-127
CPU FAST CALLS SERVICE DESCRIPTOR UNIQUE OEN RANGE: 65
STANDARD FAST CALLS SERVICE DESCRIPTOR (PSCI CALLS) UNIQUE OEN RANGE: 68

Handling an SMC

When the EL3 runtime services framework receives a Secure Monitor Call, the SMC Function ID is passed in W0 from the lower exception level (as per the SMCCC). If the calling register width is AArch32, it is invalid to invoke an SMC Function which indicates the SMC64 calling convention: such calls are ignored and return the Unknown SMC Function Identifier result code `0xFFFFFFFF` in R0/X0.

Bit[31] (fast/yielding call) and bits[29:24] (owning entity number) of the SMC Function ID are combined to index into the `rt_svc_descs_indices[]` array. The resulting value might indicate a service that has no handler, in this case the framework will also report an Unknown SMC Function ID. Otherwise, the value is used as a further index into the `rt_svc_descs[]` array to locate the required service and handler.

The service's `handle()` callback is provided with five of the SMC parameters directly, the others are saved into memory for retrieval (if needed) by the handler. The handler is also provided with an opaque `handle` for use with the supporting library for parameter retrieval, setting return values and context manipulation; and with `flags` indicating the security state of the caller. The framework finally sets up the execution stack for the handler, and invokes the services `handle()` function.

On return from the handler the result registers are populated in X0-X7 as needed before restoring the stack and CPU state and returning from the original SMC.

5.4.3 Exception Handling Framework

Please refer to the *Exception Handling Framework* document.

5.4.4 Power State Coordination Interface

TODO: Provide design walkthrough of PSCI implementation.

The PSCI v1.1 specification categorizes APIs as optional and mandatory. All the mandatory APIs in PSCI v1.1, PSCI v1.0 and in PSCI v0.2 draft specification Power State Coordination Interface PDD are implemented. The table lists the PSCI v1.1 APIs and their support in generic code.

An API implementation might have a dependency on platform code e.g. CPU_SUSPEND requires the platform to export a part of the implementation. Hence the level of support of the mandatory APIs depends upon the support exported by the platform port as well. The Juno and FVP (all variants) platforms export all the required support.

PSCI v1.1 API	Supported	Comments
`PSCI_VERSION`	Yes	The version returned is 1.1
`CPU_SUSPEND`	Yes*	
`CPU_OFF`	Yes*	
`CPU_ON`	Yes*	
`AFFINITY_INFO`	Yes	
`MIGRATE`	Yes**	
`MIGRATE_INFO_TYPE`	Yes**	
`MIGRATE_INFO_CPU`	Yes**	
`SYSTEM_OFF`	Yes*	
`SYSTEM_RESET`	Yes*	
`PSCI_FEATURES`	Yes	
`CPU_FREEZE`	No	
`CPU_DEFAULT_SUSPEND`	No	
`NODE_HW_STATE`	Yes*	
`SYSTEM_SUSPEND`	Yes*	
`PSCI_SET_SUSPEND_MODE`	No	
`PSCI_STAT_RESIDENCY`	Yes*	
`PSCI_STAT_COUNT`	Yes*	
`SYSTEM_RESET2`	Yes*	
`MEM_PROTECT`	Yes*	
`MEM_PROTECT_CHECK_RANGE`	Yes*	

*Note : These PSCI APIs require platform power management hooks to be registered with the generic PSCI code to be supported.

**Note : These PSCI APIs require appropriate Secure Payload Dispatcher hooks to be registered with the generic PSCI code to be supported.

The PSCI implementation in TF-A is a library which can be integrated with AArch64 or AArch32 EL3 Runtime Software for Armv8-A systems. A guide to integrating PSCI library with AArch32 EL3 Runtime Software can be found at *PSCI Library Integration guide for Armv8-A AArch32 systems*.

5.4.5 Secure-EL1 Payloads and Dispatchers

On a production system that includes a Trusted OS running in Secure-EL1/EL0, the Trusted OS is coupled with a companion runtime service in the BL31 firmware. This service is responsible for the initialisation of the Trusted OS and all communications with it. The Trusted OS is the BL32 stage of the boot flow in TF-A. The firmware will attempt to locate, load and execute a BL32 image.

TF-A uses a more general term for the BL32 software that runs at Secure-EL1 - the *Secure-EL1 Payload* - as it is not always a Trusted OS.

TF-A provides a Test Secure-EL1 Payload (TSP) and a Test Secure-EL1 Payload Dispatcher (TSPD) service as an example of how a Trusted OS is supported on a production system using the Runtime Services Framework. On such a system, the Test BL32 image and service are replaced by the Trusted OS and its dispatcher service. The TF-A build system expects that the dispatcher will define the build flag `NEED_BL32` to enable it to include the BL32 in the build either as a binary or to compile from source depending on whether the `BL32` build option is specified or not.

The TSP runs in Secure-EL1. It is designed to demonstrate synchronous communication with the normal-world software running in EL1/EL2. Communication is initiated by the normal-world software

- either directly through a Fast SMC (as defined in the SMCCC)

- or indirectly through a PSCI SMC. The PSCI implementation in turn informs the TSPD about the requested power management operation. This allows the TSP to prepare for or respond to the power state change

The TSPD service is responsible for.

- Initializing the TSP
- Routing requests and responses between the secure and the non-secure states during the two types of communications just described

Initializing a BL32 Image

The Secure-EL1 Payload Dispatcher (SPD) service is responsible for initializing the BL32 image. It needs access to the information passed by BL2 to BL31 to do so. This is provided by:

```
entry_point_info_t *bl31_plat_get_next_image_ep_info(uint32_t);
```

which returns a reference to the `entry_point_info` structure corresponding to the image which will be run in the specified security state. The SPD uses this API to get entry point information for the SECURE image, BL32.

In the absence of a BL32 image, BL31 passes control to the normal world bootloader image (BL33). When the BL32 image is present, it is typical that the SPD wants control to be passed to BL32 first and then later to BL33.

To do this the SPD has to register a BL32 initialization function during initialization of the SPD service. The BL32 initialization function has this prototype:

```
int32_t init(void);
```

and is registered using the `bl31_register_bl32_init()` function.

TF-A supports two approaches for the SPD to pass control to BL32 before returning through EL3 and running the non-trusted firmware (BL33):

1. In the BL32 setup function, use `bl31_set_next_image_type()` to request that the exit from `bl31_main()` is to the BL32 entrypoint in Secure-EL1. BL31 will exit to BL32 using the asynchronous method by calling `bl31_prepare_next_image_entry()` and `el3_exit()`.

   When the BL32 has completed initialization at Secure-EL1, it returns to BL31 by issuing an SMC, using a Function ID allocated to the SPD. On receipt of this SMC, the SPD service handler should switch the CPU context from trusted to normal world and use the `bl31_set_next_image_type()` and `bl31_prepare_next_image_entry()` functions to set up the initial return to the normal world firmware BL33. On return from the handler the framework will exit to EL2 and run BL33.

2. The BL32 setup function registers an initialization function using `bl31_register_bl32_init()` which provides a SPD-defined mechanism to invoke a 'world-switch synchronous call' to Secure-EL1 to run the BL32 entrypoint.

   Note: The Test SPD service included with TF-A provides one implementation of such a mechanism.

   On completion BL32 returns control to BL31 via a SMC, and on receipt the SPD service handler invokes the synchronous call return mechanism to return to the BL32 initialization function. On return from this function, `bl31_main()` will set up the return to the normal world firmware BL33 and continue the boot process in the normal world.

5.4.6 Crash Reporting in BL31

BL31 implements a scheme for reporting the processor state when an unhandled exception is encountered. The reporting mechanism attempts to preserve all the register contents and report it via a dedicated UART (PL011 console). BL31 reports the general purpose, EL3, Secure EL1 and some EL2 state registers.

A dedicated per-CPU crash stack is maintained by BL31 and this is retrieved via the per-CPU pointer cache. The implementation attempts to minimise the memory required for this feature. The file `crash_reporting.S` contains the implementation for crash reporting.

The sample crash output is shown below.

```
x0              = 0x000000002a4a0000
x1              = 0x0000000000000001
x2              = 0x0000000000000002
x3              = 0x0000000000000003
x4              = 0x0000000000000004
x5              = 0x0000000000000005
x6              = 0x0000000000000006
x7              = 0x0000000000000007
x8              = 0x0000000000000008
x9              = 0x0000000000000009
x10             = 0x0000000000000010
x11             = 0x0000000000000011
x12             = 0x0000000000000012
x13             = 0x0000000000000013
x14             = 0x0000000000000014
x15             = 0x0000000000000015
x16             = 0x0000000000000016
x17             = 0x0000000000000017
x18             = 0x0000000000000018
x19             = 0x0000000000000019
x20             = 0x0000000000000020
x21             = 0x0000000000000021
x22             = 0x0000000000000022
x23             = 0x0000000000000023
x24             = 0x0000000000000024
x25             = 0x0000000000000025
x26             = 0x0000000000000026
x27             = 0x0000000000000027
x28             = 0x0000000000000028
x29             = 0x0000000000000029
x30             = 0x0000000088000b78
scr_el3         = 0x000000000003073d
sctlr_el3       = 0x00000000b0cd183f
cptr_el3        = 0x0000000000000000
tcr_el3         = 0x000000008080351c
daif            = 0x00000000000002c0
mair_el3        = 0x00000000004404ff
spsr_el3        = 0x0000000060000349
elr_el3         = 0x0000000088000114
ttbr0_el3       = 0x0000000004018201
esr_el3         = 0x00000000be000000
far_el3         = 0x0000000000000000
spsr_el1        = 0x0000000000000000
elr_el1         = 0x0000000000000000
spsr_abt        = 0x0000000000000000
spsr_und        = 0x0000000000000000
```

(continues on next page)

(continued from previous page)

```
spsr_irq       = 0x0000000000000000
spsr_fiq       = 0x0000000000000000
sctlr_el1      = 0x0000000030d00800
actlr_el1      = 0x0000000000000000
cpacr_el1      = 0x0000000000000000
csselr_el1     = 0x0000000000000000
sp_el1         = 0x0000000000000000
esr_el1        = 0x0000000000000000
ttbr0_el1      = 0x0000000000000000
ttbr1_el1      = 0x0000000000000000
mair_el1       = 0x0000000000000000
amair_el1      = 0x0000000000000000
tcr_el1        = 0x0000000000000000
tpidr_el1      = 0x0000000000000000
tpidr_el0      = 0x0000000000000000
tpidrro_el0    = 0x0000000000000000
par_el1        = 0x0000000000000000
mpidr_el1      = 0x0000000080000000
afsr0_el1      = 0x0000000000000000
afsr1_el1      = 0x0000000000000000
contextidr_el1 = 0x0000000000000000
vbar_el1       = 0x0000000000000000
cntp_ctl_el0   = 0x0000000000000000
cntp_cval_el0  = 0x0000000000000000
cntv_ctl_el0   = 0x0000000000000000
cntv_cval_el0  = 0x0000000000000000
cntkctl_el1    = 0x0000000000000000
sp_el0         = 0x0000000004014940
isr_el1        = 0x0000000000000000
dacr32_el2     = 0x0000000000000000
ifsr32_el2     = 0x0000000000000000
icc_hppir0_el1 = 0x00000000000003ff
icc_hppir1_el1 = 0x00000000000003ff
icc_ctlr_el3   = 0x0000000000080400
gicd_ispendr regs (Offsets 0x200-0x278)
Offset                  Value
0x200:           0x0000000000000000
0x208:           0x0000000000000000
0x210:           0x0000000000000000
0x218:           0x0000000000000000
0x220:           0x0000000000000000
0x228:           0x0000000000000000
0x230:           0x0000000000000000
0x238:           0x0000000000000000
0x240:           0x0000000000000000
0x248:           0x0000000000000000
0x250:           0x0000000000000000
0x258:           0x0000000000000000
0x260:           0x0000000000000000
0x268:           0x0000000000000000
0x270:           0x0000000000000000
0x278:           0x0000000000000000
```

5.4.7 Guidelines for Reset Handlers

TF-A implements a framework that allows CPU and platform ports to perform actions very early after a CPU is released from reset in both the cold and warm boot paths. This is done by calling the `reset_handler()` function in both the BL1 and BL31 images. It in turn calls the platform and CPU specific reset handling functions.

Details for implementing a CPU specific reset handler can be found in Section 8. Details for implementing a platform specific reset handler can be found in the *Porting Guide* (see the `plat_reset_handler()` function).

When adding functionality to a reset handler, keep in mind that if a different reset handling behavior is required between the first and the subsequent invocations of the reset handling code, this should be detected at runtime. In other words, the reset handler should be able to detect whether an action has already been performed and act as appropriate. Possible courses of actions are, e.g. skip the action the second time, or undo/redo it.

5.4.8 Configuring secure interrupts

The GIC driver is responsible for performing initial configuration of secure interrupts on the platform. To this end, the platform is expected to provide the GIC driver (either GICv2 or GICv3, as selected by the platform) with the interrupt configuration during the driver initialisation.

Secure interrupt configuration are specified in an array of secure interrupt properties. In this scheme, in both GICv2 and GICv3 driver data structures, the `interrupt_props` member points to an array of interrupt properties. Each element of the array specifies the interrupt number and its attributes (priority, group, configuration). Each element of the array shall be populated by the macro `INTR_PROP_DESC()`. The macro takes the following arguments:

- 10-bit interrupt number,
- 8-bit interrupt priority,
- Interrupt type (one of `INTR_TYPE_EL3`, `INTR_TYPE_S_EL1`, `INTR_TYPE_NS`),
- Interrupt configuration (either `GIC_INTR_CFG_LEVEL` or `GIC_INTR_CFG_EDGE`).

5.4.9 CPU specific operations framework

Certain aspects of the Armv8-A architecture are implementation defined, that is, certain behaviours are not architecturally defined, but must be defined and documented by individual processor implementations. TF-A implements a framework which categorises the common implementation defined behaviours and allows a processor to export its implementation of that behaviour. The categories are:

1. Processor specific reset sequence.
2. Processor specific power down sequences.
3. Processor specific register dumping as a part of crash reporting.
4. Errata status reporting.

Each of the above categories fulfils a different requirement.

1. allows any processor specific initialization before the caches and MMU are turned on, like implementation of errata workarounds, entry into the intra-cluster coherency domain etc.
2. allows each processor to implement the power down sequence mandated in its Technical Reference Manual (TRM).
3. allows a processor to provide additional information to the developer in the event of a crash, for example Cortex-A53 has registers which can expose the data cache contents.

4. allows a processor to define a function that inspects and reports the status of all errata workarounds on that processor.

Please note that only 2. is mandated by the TRM.

The CPU specific operations framework scales to accommodate a large number of different CPUs during power down and reset handling. The platform can specify any CPU optimization it wants to enable for each CPU. It can also specify the CPU errata workarounds to be applied for each CPU type during reset handling by defining CPU errata compile time macros. Details on these macros can be found in the *Arm CPU Specific Build Macros* document.

The CPU specific operations framework depends on the `cpu_ops` structure which needs to be exported for each type of CPU in the platform. It is defined in `include/lib/cpus/aarch64/cpu_macros.S` and has the following fields : `midr`, `reset_func()`, `cpu_pwr_down_ops` (array of power down functions) and `cpu_reg_dump()`.

The CPU specific files in `lib/cpus` export a `cpu_ops` data structure with suitable handlers for that CPU. For example, `lib/cpus/aarch64/cortex_a53.S` exports the `cpu_ops` for Cortex-A53 CPU. According to the platform configuration, these CPU specific files must be included in the build by the platform makefile. The generic CPU specific operations framework code exists in `lib/cpus/aarch64/cpu_helpers.S`.

CPU specific Reset Handling

After a reset, the state of the CPU when it calls generic reset handler is: MMU turned off, both instruction and data caches turned off and not part of any coherency domain.

The BL entrypoint code first invokes the `plat_reset_handler()` to allow the platform to perform any system initialization required and any system errata workarounds that needs to be applied. The `get_cpu_ops_ptr()` reads the current CPU midr, finds the matching `cpu_ops` entry in the `cpu_ops` array and returns it. Note that only the part number and implementer fields in midr are used to find the matching `cpu_ops` entry. The `reset_func()` in the returned `cpu_ops` is then invoked which executes the required reset handling for that CPU and also any errata workarounds enabled by the platform. This function must preserve the values of general purpose registers x20 to x29.

Refer to Section “Guidelines for Reset Handlers” for general guidelines regarding placement of code in a reset handler.

CPU specific power down sequence

During the BL31 initialization sequence, the pointer to the matching `cpu_ops` entry is stored in per-CPU data by `init_cpu_ops()` so that it can be quickly retrieved during power down sequences.

Various CPU drivers register handlers to perform power down at certain power levels for that specific CPU. The PSCI service, upon receiving a power down request, determines the highest power level at which to execute power down sequence for a particular CPU. It uses the `prepare_cpu_pwr_dwn()` function to pick the right power down handler for the requested level. The function retrieves `cpu_ops` pointer member of per-CPU data, and from that, further retrieves `cpu_pwr_down_ops` array, and indexes into the required level. If the requested power level is higher than what a CPU driver supports, the handler registered for highest level is invoked.

At runtime the platform hooks for power down are invoked by the PSCI service to perform platform specific operations during a power down sequence, for example turning off CCI coherency during a cluster power down.

CPU specific register reporting during crash

If the crash reporting is enabled in BL31, when a crash occurs, the crash reporting framework calls `do_cpu_reg_dump` which retrieves the matching `cpu_ops` using `get_cpu_ops_ptr()` function. The `cpu_reg_dump()` in `cpu_ops` is invoked, which then returns the CPU specific register values to be reported and a pointer to the ASCII list of register names in a format expected by the crash reporting framework.

CPU errata status reporting

Errata workarounds for CPUs supported in TF-A are applied during both cold and warm boots, shortly after reset. Individual Errata workarounds are enabled as build options. Some errata workarounds have potential run-time implications; therefore some are enabled by default, others not. Platform ports shall override build options to enable or disable errata as appropriate. The CPU drivers take care of applying errata workarounds that are enabled and applicable to a given CPU. Refer to *CPU Errata Workarounds* for more information.

Functions in CPU drivers that apply errata workaround must follow the conventions listed below.

The errata workaround must be authored as two separate functions:

- One that checks for errata. This function must determine whether that errata applies to the current CPU. Typically this involves matching the current CPUs revision and variant against a value that's known to be affected by the errata. If the function determines that the errata applies to this CPU, it must return `ERRATA_APPLIES`; otherwise, it must return `ERRATA_NOT_APPLIES`. The utility functions `cpu_get_rev_var` and `cpu_rev_var_ls` functions may come in handy for this purpose.

For an errata identified as `E`, the check function must be named `check_errata_E`.

This function will be invoked at different times, both from assembly and from C run time. Therefore it must follow AAPCS, and must not use stack.

- Another one that applies the errata workaround. This function would call the check function described above, and applies errata workaround if required.

CPU drivers that apply errata workaround can optionally implement an assembly function that report the status of errata workarounds pertaining to that CPU. For a driver that registers the CPU, for example, `cpux` via `declare_cpu_ops` macro, the errata reporting function, if it exists, must be named `cpux_errata_report`. This function will always be called with MMU enabled; it must follow AAPCS and may use stack.

In a debug build of TF-A, on a CPU that comes out of reset, both BL1 and the runtime firmware (BL31 in AArch64, and BL32 in AArch32) will invoke errata status reporting function, if one exists, for that type of CPU.

To report the status of each errata workaround, the function shall use the assembler macro `report_errata`, passing it:

- The build option that enables the errata;
- The name of the CPU: this must be the same identifier that CPU driver registered itself with, using `declare_cpu_ops`;
- And the errata identifier: the identifier must match what's used in the errata's check function described above.

The errata status reporting function will be called once per CPU type/errata combination during the software's active life time.

It's expected that whenever an errata workaround is submitted to TF-A, the errata reporting function is appropriately extended to report its status as well.

Reporting the status of errata workaround is for informational purpose only; it has no functional significance.

5.4.10 Memory layout of BL images

Each bootloader image can be divided in 2 parts:

- the static contents of the image. These are data actually stored in the binary on the disk. In the ELF terminology, they are called `PROGBITS` sections;
- the run-time contents of the image. These are data that don't occupy any space in the binary on the disk. The ELF binary just contains some metadata indicating where these data will be stored at run-time and the corresponding sections need to be allocated and initialized at run-time. In the ELF terminology, they are called `NOBITS` sections.

All PROGBITS sections are grouped together at the beginning of the image, followed by all NOBITS sections. This is true for all TF-A images and it is governed by the linker scripts. This ensures that the raw binary images are as small as possible. If a NOBITS section was inserted in between PROGBITS sections then the resulting binary file would contain zero bytes in place of this NOBITS section, making the image unnecessarily bigger. Smaller images allow faster loading from the FIP to the main memory.

For BL31, a platform can specify an alternate location for NOBITS sections (other than immediately following PROGBITS sections) by setting `SEPARATE_NOBITS_REGION` to 1 and defining `BL31_NOBITS_BASE` and `BL31_NOBITS_LIMIT`.

Linker scripts and symbols

Each bootloader stage image layout is described by its own linker script. The linker scripts export some symbols into the program symbol table. Their values correspond to particular addresses. TF-A code can refer to these symbols to figure out the image memory layout.

Linker symbols follow the following naming convention in TF-A.

- `__<SECTION>_START__`

  Start address of a given section named `<SECTION>`.

- `__<SECTION>_END__`

  End address of a given section named `<SECTION>`. If there is an alignment constraint on the section's end address then `__<SECTION>_END__` corresponds to the end address of the section's actual contents, rounded up to the right boundary. Refer to the value of `__<SECTION>_UNALIGNED_END__` to know the actual end address of the section's contents.

- `__<SECTION>_UNALIGNED_END__`

  End address of a given section named `<SECTION>` without any padding or rounding up due to some alignment constraint.

- `__<SECTION>_SIZE__`

  Size (in bytes) of a given section named `<SECTION>`. If there is an alignment constraint on the section's end address then `__<SECTION>_SIZE__` corresponds to the size of the section's actual contents, rounded up to the right boundary. In other words, `__<SECTION>_SIZE__ = __<SECTION>_END__ - _<SECTION>_START__`. Refer to the value of `__<SECTION>_UNALIGNED_SIZE__` to know the actual size of the section's contents.

- `__<SECTION>_UNALIGNED_SIZE__`

  Size (in bytes) of a given section named `<SECTION>` without any padding or rounding up due to some alignment constraint. In other words, `__<SECTION>_UNALIGNED_SIZE__ = __<SECTION>_UNALIGNED_END__ - __<SECTION>_START__`.

Some of the linker symbols are mandatory as TF-A code relies on them to be defined. They are listed in the following subsections. Some of them must be provided for each bootloader stage and some are specific to a given bootloader stage.

The linker scripts define some extra, optional symbols. They are not actually used by any code but they help in understanding the bootloader images' memory layout as they are easy to spot in the link map files.

Common linker symbols

All BL images share the following requirements:

- The BSS section must be zero-initialised before executing any C code.
- The coherent memory section (if enabled) must be zero-initialised as well.
- The MMU setup code needs to know the extents of the coherent and read-only memory regions to set the right memory attributes. When `SEPARATE_CODE_AND_RODATA=1`, it needs to know more specifically how the read-only memory region is divided between code and data.

The following linker symbols are defined for this purpose:

- `__BSS_START__`
- `__BSS_SIZE__`
- `__COHERENT_RAM_START__` Must be aligned on a page-size boundary.
- `__COHERENT_RAM_END__` Must be aligned on a page-size boundary.
- `__COHERENT_RAM_UNALIGNED_SIZE__`
- `__RO_START__`
- `__RO_END__`
- `__TEXT_START__`
- `__TEXT_END__`
- `__RODATA_START__`
- `__RODATA_END__`

BL1's linker symbols

BL1 being the ROM image, it has additional requirements. BL1 resides in ROM and it is entirely executed in place but it needs some read-write memory for its mutable data. Its `.data` section (i.e. its allocated read-write data) must be relocated from ROM to RAM before executing any C code.

The following additional linker symbols are defined for BL1:

- `__BL1_ROM_END__` End address of BL1's ROM contents, covering its code and `.data` section in ROM.
- `__DATA_ROM_START__` Start address of the `.data` section in ROM. Must be aligned on a 16-byte boundary.
- `__DATA_RAM_START__` Address in RAM where the `.data` section should be copied over. Must be aligned on a 16-byte boundary.
- `__DATA_SIZE__` Size of the `.data` section (in ROM or RAM).
- `__BL1_RAM_START__` Start address of BL1 read-write data.
- `__BL1_RAM_END__` End address of BL1 read-write data.

How to choose the right base addresses for each bootloader stage image

There is currently no support for dynamic image loading in TF-A. This means that all bootloader images need to be linked against their ultimate runtime locations and the base addresses of each image must be chosen carefully such that images don't overlap each other in an undesired way. As the code grows, the base addresses might need adjustments to cope with the new memory layout.

The memory layout is completely specific to the platform and so there is no general recipe for choosing the right base addresses for each bootloader image. However, there are tools to aid in understanding the memory layout. These are the link map files: `build/<platform>/<build-type>/bl<x>/bl<x>.map`, with `<x>` being the stage bootloader. They provide a detailed view of the memory usage of each image. Among other useful information, they provide the end address of each image.

- `bl1.map` link map file provides `__BL1_RAM_END__` address.
- `bl2.map` link map file provides `__BL2_END__` address.
- `bl31.map` link map file provides `__BL31_END__` address.
- `bl32.map` link map file provides `__BL32_END__` address.

For each bootloader image, the platform code must provide its start address as well as a limit address that it must not overstep. The latter is used in the linker scripts to check that the image doesn't grow past that address. If that happens, the linker will issue a message similar to the following:

```
aarch64-none-elf-ld: BLx has exceeded its limit.
```

Additionally, if the platform memory layout implies some image overlaying like on FVP, BL31 and TSP need to know the limit address that their PROGBITS sections must not overstep. The platform code must provide those.

TF-A does not provide any mechanism to verify at boot time that the memory to load a new image is free to prevent overwriting a previously loaded image. The platform must specify the memory available in the system for all the relevant BL images to be loaded.

For example, in the case of BL1 loading BL2, `bl1_plat_sec_mem_layout()` will return the region defined by the platform where BL1 intends to load BL2. The `load_image()` function performs bounds check for the image size based on the base and maximum image size provided by the platforms. Platforms must take this behaviour into account when defining the base/size for each of the images.

Memory layout on Arm development platforms

The following list describes the memory layout on the Arm development platforms:

- A 4KB page of shared memory is used for communication between Trusted Firmware and the platform's power controller. This is located at the base of Trusted SRAM. The amount of Trusted SRAM available to load the bootloader images is reduced by the size of the shared memory.

  The shared memory is used to store the CPUs' entrypoint mailbox. On Juno, this is also used for the MHU payload when passing messages to and from the SCP.
- Another 4 KB page is reserved for passing memory layout between BL1 and BL2 and also the dynamic firmware configurations.
- On FVP, BL1 is originally sitting in the Trusted ROM at address `0x0`. On Juno, BL1 resides in flash memory at address `0x0BEC0000`. BL1 read-write data are relocated to the top of Trusted SRAM at runtime.
- BL2 is loaded below BL1 RW
- EL3 Runtime Software, BL31 for AArch64 and BL32 for AArch32 (e.g. SP_MIN), is loaded at the top of the Trusted SRAM, such that its NOBITS sections will overwrite BL1 R/W data and BL2. This implies that BL1

global variables remain valid only until execution reaches the EL3 Runtime Software entry point during a cold boot.

- On Juno, SCP_BL2 is loaded temporarily into the EL3 Runtime Software memory region and transferred to the SCP before being overwritten by EL3 Runtime Software.
- BL32 (for AArch64) can be loaded in one of the following locations:
  - Trusted SRAM
  - Trusted DRAM (FVP only)
  - Secure region of DRAM (top 16MB of DRAM configured by the TrustZone controller)

  When BL32 (for AArch64) is loaded into Trusted SRAM, it is loaded below BL31.

The location of the BL32 image will result in different memory maps. This is illustrated for both FVP and Juno in the following diagrams, using the TSP as an example.

Note: Loading the BL32 image in TZC secured DRAM doesn't change the memory layout of the other images in Trusted SRAM.

CONFIG section in memory layouts shown below contains:

```
+--------------------+
|bl2_mem_params_descs|
|--------------------|
|     fw_configs     |
+--------------------+
```

`bl2_mem_params_descs` contains parameters passed from BL2 to next the BL image during boot.

`fw_configs` includes soc_fw_config, tos_fw_config, tb_fw_config and fw_config.

FVP with TSP in Trusted SRAM with firmware configs : (These diagrams only cover the AArch64 case)

```
               DRAM
0xffffffff +----------+
           :          :
           |----------|
           |HW_CONFIG |
0x83000000 |----------|  (non-secure)
           |          |
0x80000000 +----------+

           Trusted SRAM
0x04040000 +----------+  loaded by BL2  +----------------+
           | BL1 (rw) |  <<<<<<<<<<<<<  |                |
           |----------|  <<<<<<<<<<<<<  |  BL31 NOBITS   |
           |   BL2    |  <<<<<<<<<<<<<  |                |
           |----------|  <<<<<<<<<<<<<  |----------------|
           |          |  <<<<<<<<<<<<<  | BL31 PROGBITS  |
           |          |  <<<<<<<<<<<<<  |----------------|
           |          |  <<<<<<<<<<<<<  |     BL32       |
0x04003000 +----------+                 +----------------+
           |  CONFIG  |
0x04001000 +----------+
           |  Shared  |
0x04000000 +----------+
```

(continues on next page)

(continued from previous page)

```
           Trusted ROM
0x04000000 +----------+
           | BL1 (ro) |
0x00000000 +----------+
```

FVP with TSP in Trusted DRAM with firmware configs (default option):

```
                 DRAM
0xffffffff +---------------+
           :               :
           |---------------|
           |  HW_CONFIG    |
0x83000000 |---------------|  (non-secure)
           |               |
0x80000000 +---------------+

            Trusted DRAM
0x08000000 +---------------+
           |     BL32      |
0x06000000 +---------------+

             Trusted SRAM
0x04040000 +---------------+  loaded by BL2  +-----------------+
           |   BL1 (rw)    |  <<<<<<<<<<<<<  |                 |
           |---------------|  <<<<<<<<<<<<<  |  BL31 NOBITS    |
           |     BL2       |  <<<<<<<<<<<<<  |                 |
           |---------------|  <<<<<<<<<<<<<  |-----------------|
           |               |  <<<<<<<<<<<<<  | BL31 PROGBITS   |
           |               |                 +-----------------+
0x04003000 +---------------+
           |    CONFIG     |
0x04001000 +---------------+
           |    Shared     |
0x04000000 +---------------+

             Trusted ROM
0x04000000 +---------------+
           |   BL1 (ro)    |
0x00000000 +---------------+
```

FVP with TSP in TZC-Secured DRAM with firmware configs :

```
               DRAM
0xffffffff +----------+
           |  BL32    |  (secure)
0xff000000 +----------+
           |          |
           |----------|
           |HW_CONFIG |
0x83000000 |----------|  (non-secure)
           |          |
0x80000000 +----------+

           Trusted SRAM
0x04040000 +----------+  loaded by BL2  +-----------------+
           | BL1 (rw) |  <<<<<<<<<<<<<  |                 |
```

(continues on next page)

(continued from previous page)

```
            |----------|  <<<<<<<<<<<<<  |  BL31 NOBITS   |
            |   BL2    |  <<<<<<<<<<<<<  |                |
            |----------|  <<<<<<<<<<<<<  |----------------|
            |          |  <<<<<<<<<<<<<  | BL31 PROGBITS  |
            |          |                 +----------------+
0x04003000 +----------+
            |  CONFIG  |
0x04001000 +----------+
            |  Shared  |
0x04000000 +----------+

           Trusted ROM
0x04000000 +----------+
            | BL1 (ro) |
0x00000000 +----------+
```

Juno with BL32 in Trusted SRAM :

```
              Flash0
0x0C000000 +----------+
           :          :
0x0BED0000 |----------|
           | BL1 (ro) |
0x0BEC0000 |----------|
           :          :
0x08000000 +----------+                  BL31 is loaded
                                         after SCP_BL2 has
           Trusted SRAM                  been sent to SCP
0x04040000 +----------+  loaded by BL2  +----------------+
           | BL1 (rw) |  <<<<<<<<<<<<<  |                |
           |----------|  <<<<<<<<<<<<<  |  BL31 NOBITS   |
           |   BL2    |  <<<<<<<<<<<<<  |                |
           |----------|  <<<<<<<<<<<<<  |----------------|
           | SCP_BL2  |  <<<<<<<<<<<<<  | BL31 PROGBITS  |
           |          |  <<<<<<<<<<<<<  |----------------|
           |          |  <<<<<<<<<<<<<  |     BL32       |
           |          |                 +----------------+
           |          |
0x04001000 +----------+
           |   MHU    |
0x04000000 +----------+
```

Juno with BL32 in TZC-secured DRAM :

```
               DRAM
0xFFE00000 +----------+
           |  BL32    |  (secure)
0xFF000000 |----------|
           |          |
           :          :  (non-secure)
           |          |
0x80000000 +----------+

              Flash0
0x0C000000 +----------+
           :          :
0x0BED0000 |----------|
```

(continues on next page)

(continued from previous page)

```
           | BL1 (ro) |
0x0BEC0000 |----------|
           :          :
0x08000000 +----------+                 BL31 is loaded
                                        after SCP_BL2 has
           Trusted SRAM                 been sent to SCP
0x04040000 +----------+  loaded by BL2  +----------------+
           | BL1 (rw) |  <<<<<<<<<<<<<  |                |
           |----------|  <<<<<<<<<<<<<  |  BL31 NOBITS   |
           |   BL2    |  <<<<<<<<<<<<<  |                |
           |----------|  <<<<<<<<<<<<<  |----------------|
           | SCP_BL2  |  <<<<<<<<<<<<<  | BL31 PROGBITS  |
           |          |                 +----------------+
0x04001000 +----------+
           |   MHU    |
0x04000000 +----------+
```

5.4.11 Firmware Image Package (FIP)

Using a Firmware Image Package (FIP) allows for packing bootloader images (and potentially other payloads) into a single archive that can be loaded by TF-A from non-volatile platform storage. A driver to load images from a FIP has been added to the storage layer and allows a package to be read from supported platform storage. A tool to create Firmware Image Packages is also provided and described below.

Firmware Image Package layout

The FIP layout consists of a table of contents (ToC) followed by payload data. The ToC itself has a header followed by one or more table entries. The ToC is terminated by an end marker entry, and since the size of the ToC is 0 bytes, the offset equals the total size of the FIP file. All ToC entries describe some payload data that has been appended to the end of the binary package. With the information provided in the ToC entry the corresponding payload data can be retrieved.

```
------------------
| ToC Header     |
|----------------|
| ToC Entry 0    |
|----------------|
| ToC Entry 1    |
|----------------|
| ToC End Marker |
|----------------|
|                |
|     Data 0     |
|                |
|----------------|
|                |
|     Data 1     |
|                |
------------------
```

The ToC header and entry formats are described in the header file `include/tools_share/firmware_image_package.h`. This file is used by both the tool and TF-A.

The ToC header has the following fields:

```
`name`: The name of the ToC. This is currently used to validate the header.
`serial_number`: A non-zero number provided by the creation tool
`flags`: Flags associated with this data.
    Bits 0-31: Reserved
    Bits 32-47: Platform defined
    Bits 48-63: Reserved
```

A ToC entry has the following fields:

```
`uuid`: All files are referred to by a pre-defined Universally Unique
    IDentifier [UUID] . The UUIDs are defined in
    `include/tools_share/firmware_image_package.h`. The platform translates
    the requested image name into the corresponding UUID when accessing the
    package.
`offset_address`: The offset address at which the corresponding payload data
    can be found. The offset is calculated from the ToC base address.
`size`: The size of the corresponding payload data in bytes.
`flags`: Flags associated with this entry. None are yet defined.
```

Firmware Image Package creation tool

The FIP creation tool can be used to pack specified images into a binary package that can be loaded by TF-A from platform storage. The tool currently only supports packing bootloader images. Additional image definitions can be added to the tool as required.

The tool can be found in `tools/fiptool`.

Loading from a Firmware Image Package (FIP)

The Firmware Image Package (FIP) driver can load images from a binary package on non-volatile platform storage. For the Arm development platforms, this is currently NOR FLASH.

Bootloader images are loaded according to the platform policy as specified by the function `plat_get_image_source()`. For the Arm development platforms, this means the platform will attempt to load images from a Firmware Image Package located at the start of NOR FLASH0.

The Arm development platforms' policy is to only allow loading of a known set of images. The platform policy can be modified to allow additional images.

5.4.12 Use of coherent memory in TF-A

There might be loss of coherency when physical memory with mismatched shareability, cacheability and memory attributes is accessed by multiple CPUs (refer to section B2.9 of Arm ARM for more details). This possibility occurs in TF-A during power up/down sequences when coherency, MMU and caches are turned on/off incrementally.

TF-A defines coherent memory as a region of memory with Device nGnRE attributes in the translation tables. The translation granule size in TF-A is 4KB. This is the smallest possible size of the coherent memory region.

By default, all data structures which are susceptible to accesses with mismatched attributes from various CPUs are allocated in a coherent memory region (refer to section 2.1 of *Porting Guide*). The coherent memory region accesses are Outer Shareable, non-cacheable and they can be accessed with the Device nGnRE attributes when the MMU is turned on. Hence, at the expense of at least an extra page of memory, TF-A is able to work around coherency issues due to mismatched memory attributes.

The alternative to the above approach is to allocate the susceptible data structures in Normal WriteBack WriteAllocate Inner shareable memory. This approach requires the data structures to be designed so that it is possible to work around the issue of mismatched memory attributes by performing software cache maintenance on them.

Disabling the use of coherent memory in TF-A

It might be desirable to avoid the cost of allocating coherent memory on platforms which are memory constrained. TF-A enables inclusion of coherent memory in firmware images through the build flag `USE_COHERENT_MEM`. This flag is enabled by default. It can be disabled to choose the second approach described above.

The below sections analyze the data structures allocated in the coherent memory region and the changes required to allocate them in normal memory.

Coherent memory usage in PSCI implementation

The `psci_non_cpu_pd_nodes` data structure stores the platform's power domain tree information for state management of power domains. By default, this data structure is allocated in the coherent memory region in TF-A because it can be accessed by multiple CPUs, either with caches enabled or disabled.

```
typedef struct non_cpu_pwr_domain_node {
    /*
     * Index of the first CPU power domain node level 0 which has this node
     * as its parent.
     */
    unsigned int cpu_start_idx;

    /*
     * Number of CPU power domains which are siblings of the domain indexed
     * by 'cpu_start_idx' i.e. all the domains in the range 'cpu_start_idx
     * -> cpu_start_idx + ncpus' have this node as their parent.
     */
    unsigned int ncpus;

    /*
     * Index of the parent power domain node.
     */
    unsigned int parent_node;

    plat_local_state_t local_state;

    unsigned char level;

    /* For indexing the psci_lock array*/
    unsigned char lock_index;
} non_cpu_pd_node_t;
```

In order to move this data structure to normal memory, the use of each of its fields must be analyzed. Fields like `cpu_start_idx`, `ncpus`, `parent_node level` and `lock_index` are only written once during cold boot. Hence removing them from coherent memory involves only doing a clean and invalidate of the cache lines after these fields are written.

The field `local_state` can be concurrently accessed by multiple CPUs in different cache states. A Lamport's Bakery lock `psci_locks` is used to ensure mutual exclusion to this field and a clean and invalidate is needed after it is written.

Bakery lock data

The bakery lock data structure `bakery_lock_t` is allocated in coherent memory and is accessed by multiple CPUs with mismatched attributes. `bakery_lock_t` is defined as follows:

```
typedef struct bakery_lock {
    /*
     * The lock_data is a bit-field of 2 members:
     * Bit[0]       : choosing. This field is set when the CPU is
     *                choosing its bakery number.
     * Bits[1 - 15] : number. This is the bakery number allocated.
     */
    volatile uint16_t lock_data[BAKERY_LOCK_MAX_CPUS];
} bakery_lock_t;
```

It is a characteristic of Lamport's Bakery algorithm that the volatile per-CPU fields can be read by all CPUs but only written to by the owning CPU.

Depending upon the data cache line size, the per-CPU fields of the `bakery_lock_t` structure for multiple CPUs may exist on a single cache line. These per-CPU fields can be read and written during lock contention by multiple CPUs with mismatched memory attributes. Since these fields are a part of the lock implementation, they do not have access to any other locking primitive to safeguard against the resulting coherency issues. As a result, simple software cache maintenance is not enough to allocate them in coherent memory. Consider the following example.

CPU0 updates its per-CPU field with data cache enabled. This write updates a local cache line which contains a copy of the fields for other CPUs as well. Now CPU1 updates its per-CPU field of the `bakery_lock_t` structure with data cache disabled. CPU1 then issues a DCIVAC operation to invalidate any stale copies of its field in any other cache line in the system. This operation will invalidate the update made by CPU0 as well.

To use bakery locks when `USE_COHERENT_MEM` is disabled, the lock data structure has been redesigned. The changes utilise the characteristic of Lamport's Bakery algorithm mentioned earlier. The bakery_lock structure only allocates the memory for a single CPU. The macro `DEFINE_BAKERY_LOCK` allocates all the bakery locks needed for a CPU into a section `bakery_lock`. The linker allocates the memory for other cores by using the total size allocated for the bakery_lock section and multiplying it with (PLATFORM_CORE_COUNT - 1). This enables software to perform software cache maintenance on the lock data structure without running into coherency issues associated with mismatched attributes.

The bakery lock data structure `bakery_info_t` is defined for use when `USE_COHERENT_MEM` is disabled as follows:

```
typedef struct bakery_info {
    /*
     * The lock_data is a bit-field of 2 members:
     * Bit[0]       : choosing. This field is set when the CPU is
     *                choosing its bakery number.
     * Bits[1 - 15] : number. This is the bakery number allocated.
     */
     volatile uint16_t lock_data;
} bakery_info_t;
```

The `bakery_info_t` represents a single per-CPU field of one lock and the combination of corresponding `bakery_info_t` structures for all CPUs in the system represents the complete bakery lock. The view in memory for a system with n bakery locks are:

```
bakery_lock section start
|----------------|
| `bakery_info_t`| <-- Lock_0 per-CPU field
|    Lock_0      |     for CPU0
```

(continues on next page)

(continued from previous page)

```
|----------------|
| `bakery_info_t`| <-- Lock_1 per-CPU field
|    Lock_1      |     for CPU0
|----------------|
| ....           |
|----------------|
| `bakery_info_t`| <-- Lock_N per-CPU field
|    Lock_N      |     for CPU0
------------------
|    XXXXX       |
| Padding to     |
| next Cache WB  | <--- Calculate PERCPU_BAKERY_LOCK_SIZE, allocate
|  Granule       |       continuous memory for remaining CPUs.
------------------
| `bakery_info_t`| <-- Lock_0 per-CPU field
|    Lock_0      |     for CPU1
|----------------|
| `bakery_info_t`| <-- Lock_1 per-CPU field
|    Lock_1      |     for CPU1
|----------------|
| ....           |
|----------------|
| `bakery_info_t`| <-- Lock_N per-CPU field
|    Lock_N      |     for CPU1
------------------
|    XXXXX       |
| Padding to     |
| next Cache WB  |
|  Granule       |
------------------
```

Consider a system of 2 CPUs with 'N' bakery locks as shown above. For an operation on Lock_N, the corresponding `bakery_info_t` in both CPU0 and CPU1 `bakery_lock` section need to be fetched and appropriate cache operations need to be performed for each access.

On Arm Platforms, bakery locks are used in psci (`psci_locks`) and power controller driver (`arm_lock`).

Non Functional Impact of removing coherent memory

Removal of the coherent memory region leads to the additional software overhead of performing cache maintenance for the affected data structures. However, since the memory where the data structures are allocated is cacheable, the overhead is mostly mitigated by an increase in performance.

There is however a performance impact for bakery locks, due to:

- Additional cache maintenance operations, and
- Multiple cache line reads for each lock operation, since the bakery locks for each CPU are distributed across different cache lines.

The implementation has been optimized to minimize this additional overhead. Measurements indicate that when bakery locks are allocated in Normal memory, the minimum latency of acquiring a lock is on an average 3-4 micro seconds whereas in Device memory the same is 2 micro seconds. The measurements were done on the Juno Arm development platform.

As mentioned earlier, almost a page of memory can be saved by disabling `USE_COHERENT_MEM`. Each platform needs to consider these trade-offs to decide whether coherent memory should be used. If a platform disables USE_COHERENT_MEM and needs to use bakery locks in the porting layer, it can optionally define macro PLAT_PERCPU_BAKERY_LOCK_SIZE (see the *Porting Guide*). Refer to the reference platform code for examples.

5.4.13 Isolating code and read-only data on separate memory pages

In the Armv8-A VMSA, translation table entries include fields that define the properties of the target memory region, such as its access permissions. The smallest unit of memory that can be addressed by a translation table entry is a memory page. Therefore, if software needs to set different permissions on two memory regions then it needs to map them using different memory pages.

The default memory layout for each BL image is as follows:

```
|        ...        |
+-------------------+
|  Read-write data  |
+-------------------+ Page boundary
|     <Padding>     |
+-------------------+
| Exception vectors |
+-------------------+ 2 KB boundary
|     <Padding>     |
+-------------------+
|  Read-only data   |
+-------------------+
|       Code        |
+-------------------+ BLx_BASE
```

Note: The 2KB alignment for the exception vectors is an architectural requirement.

The read-write data start on a new memory page so that they can be mapped with read-write permissions, whereas the code and read-only data below are configured as read-only.

However, the read-only data are not aligned on a page boundary. They are contiguous to the code. Therefore, the end of the code section and the beginning of the read-only data one might share a memory page. This forces both to be mapped with the same memory attributes. As the code needs to be executable, this means that the read-only data stored on the same memory page as the code are executable as well. This could potentially be exploited as part of a security attack.

TF provides the build flag SEPARATE_CODE_AND_RODATA to isolate the code and read-only data on separate memory pages. This in turn allows independent control of the access permissions for the code and read-only data. In this case, platform code gets a finer-grained view of the image layout and can appropriately map the code region as executable and the read-only data as execute-never.

This has an impact on memory footprint, as padding bytes need to be introduced between the code and read-only data to ensure the segregation of the two. To limit the memory cost, this flag also changes the memory layout such that the code and exception vectors are now contiguous, like so:

```
|        ...        |
+-------------------+
|  Read-write data  |
+-------------------+ Page boundary
|     <Padding>     |
+-------------------+
|  Read-only data   |
```

(continues on next page)

(continued from previous page)

```
+--------------------+ Page boundary
|     <Padding>      |
+--------------------+
| Exception vectors  |
+--------------------+ 2 KB boundary
|     <Padding>      |
+--------------------+
|       Code         |
+--------------------+ BLx_BASE
```

With this more condensed memory layout, the separation of read-only data will add zero or one page to the memory footprint of each BL image. Each platform should consider the trade-off between memory footprint and security.

This build flag is disabled by default, minimising memory footprint. On Arm platforms, it is enabled.

5.4.14 Publish and Subscribe Framework

The Publish and Subscribe Framework allows EL3 components to define and publish events, to which other EL3 components can subscribe.

The following macros are provided by the framework:

- `REGISTER_PUBSUB_EVENT(event)`: Defines an event, and takes one argument, the event name, which must be a valid C identifier. All calls to `REGISTER_PUBSUB_EVENT` macro must be placed in the file `pubsub_events.h`.
- `PUBLISH_EVENT_ARG(event, arg)`: Publishes a defined event, by iterating subscribed handlers and calling them in turn. The handlers will be passed the parameter `arg`. The expected use-case is to broadcast an event.
- `PUBLISH_EVENT(event)`: Like `PUBLISH_EVENT_ARG`, except that the value `NULL` is passed to subscribed handlers.
- `SUBSCRIBE_TO_EVENT(event, handler)`: Registers the `handler` to subscribe to `event`. The handler will be executed whenever the `event` is published.
- `for_each_subscriber(event, subscriber)`: Iterates through all handlers subscribed for `event`. `subscriber` must be a local variable of type `pubsub_cb_t *`, and will point to each subscribed handler in turn during iteration. This macro can be used for those patterns that none of the `PUBLISH_EVENT_*()` macros cover.

Publishing an event that wasn't defined using `REGISTER_PUBSUB_EVENT` will result in build error. Subscribing to an undefined event however won't.

Subscribed handlers must be of type `pubsub_cb_t`, with following function signature:

```
typedef void* (*pubsub_cb_t)(const void *arg);
```

There may be arbitrary number of handlers registered to the same event. The order in which subscribed handlers are notified when that event is published is not defined. Subscribed handlers may be executed in any order; handlers should not assume any relative ordering amongst them.

Publishing an event on a PE will result in subscribed handlers executing on that PE only; it won't cause handlers to execute on a different PE.

Note that publishing an event on a PE blocks until all the subscribed handlers finish executing on the PE.

TF-A generic code publishes and subscribes to some events within. Platform ports are discouraged from subscribing to them. These events may be withdrawn, renamed, or have their semantics altered in the future. Platforms may however register, publish, and subscribe to platform-specific events.

Publish and Subscribe Example

A publisher that wants to publish event `foo` would:

- Define the event `foo` in the `pubsub_events.h`.

```
REGISTER_PUBSUB_EVENT(foo);
```

- Depending on the nature of event, use one of `PUBLISH_EVENT_*()` macros to publish the event at the appropriate path and time of execution.

A subscriber that wants to subscribe to event `foo` published above would implement:

```
void *foo_handler(const void *arg)
{
     void *result;

     /* Do handling ... */

     return result;
}

SUBSCRIBE_TO_EVENT(foo, foo_handler);
```

Reclaiming the BL31 initialization code

A significant amount of the code used for the initialization of BL31 is never needed again after boot time. In order to reduce the runtime memory footprint, the memory used for this code can be reclaimed after initialization has finished and be used for runtime data.

The build option `RECLAIM_INIT_CODE` can be set to mark this boot time code with a `.text.init.*` attribute which can be filtered and placed suitably within the BL image for later reclamation by the platform. The platform can specify the filter and the memory region for this init section in BL31 via the plat.ld.S linker script. For example, on the FVP, this section is placed overlapping the secondary CPU stacks so that after the cold boot is done, this memory can be reclaimed for the stacks. The init memory section is initially mapped with `RO`, `EXECUTE` attributes. After BL31 initialization has completed, the FVP changes the attributes of this section to `RW`, `EXECUTE_NEVER` allowing it to be used for runtime data. The memory attributes are changed within the `bl31_plat_runtime_setup` platform hook. The init section section can be reclaimed for any data which is accessed after cold boot initialization and it is upto the platform to make the decision.

5.4.15 Performance Measurement Framework

The Performance Measurement Framework (PMF) facilitates collection of timestamps by registered services and provides interfaces to retrieve them from within TF-A. A platform can choose to expose appropriate SMCs to retrieve these collected timestamps.

By default, the global physical counter is used for the timestamp value and is read via `CNTPCT_EL0`. The framework allows to retrieve timestamps captured by other CPUs.

Timestamp identifier format

A PMF timestamp is uniquely identified across the system via the timestamp ID or `tid`. The `tid` is composed as follows:

```
Bits 0-7: The local timestamp identifier.
Bits 8-9: Reserved.
Bits 10-15: The service identifier.
Bits 16-31: Reserved.
```

1. The service identifier. Each PMF service is identified by a service name and a service identifier. Both the service name and identifier are unique within the system as a whole.
2. The local timestamp identifier. This identifier is unique within a given service.

Registering a PMF service

To register a PMF service, the `PMF_REGISTER_SERVICE()` macro from `pmf.h` is used. The arguments required are the service name, the service ID, the total number of local timestamps to be captured and a set of flags.

The `flags` field can be specified as a bitwise-OR of the following values:

```
PMF_STORE_ENABLE: The timestamp is stored in memory for later retrieval.
PMF_DUMP_ENABLE: The timestamp is dumped on the serial console.
```

The `PMF_REGISTER_SERVICE()` reserves memory to store captured timestamps in a PMF specific linker section at build time. Additionally, it defines necessary functions to capture and retrieve a particular timestamp for the given service at runtime.

The macro `PMF_REGISTER_SERVICE()` only enables capturing PMF timestamps from within TF-A. In order to retrieve timestamps from outside of TF-A, the `PMF_REGISTER_SERVICE_SMC()` macro must be used instead. This macro accepts the same set of arguments as the `PMF_REGISTER_SERVICE()` macro but additionally supports retrieving timestamps using SMCs.

Capturing a timestamp

PMF timestamps are stored in a per-service timestamp region. On a system with multiple CPUs, each timestamp is captured and stored in a per-CPU cache line aligned memory region.

Having registered the service, the `PMF_CAPTURE_TIMESTAMP()` macro can be used to capture a timestamp at the location where it is used. The macro takes the service name, a local timestamp identifier and a flag as arguments.

The `flags` field argument can be zero, or `PMF_CACHE_MAINT` which instructs PMF to do cache maintenance following the capture. Cache maintenance is required if any of the service's timestamps are captured with data cache disabled.

To capture a timestamp in assembly code, the caller should use `pmf_calc_timestamp_addr` macro (defined in `pmf_asm_macros.S`) to calculate the address of where the timestamp would be stored. The caller should then read `CNTPCT_EL0` register to obtain the timestamp and store it at the determined address for later retrieval.

Retrieving a timestamp

From within TF-A, timestamps for individual CPUs can be retrieved using either `PMF_GET_TIMESTAMP_BY_MPIDR()` or `PMF_GET_TIMESTAMP_BY_INDEX()` macros. These macros accept the CPU's MPIDR value, or its ordinal position respectively.

From outside TF-A, timestamps for individual CPUs can be retrieved by calling into `pmf_smc_handler()`.

```
Interface : pmf_smc_handler()
Argument  : unsigned int smc_fid, u_register_t x1,
            u_register_t x2, u_register_t x3,
            u_register_t x4, void *cookie,
            void *handle, u_register_t flags
Return    : uintptr_t

smc_fid: Holds the SMC identifier which is either `PMF_SMC_GET_TIMESTAMP_32`
    when the caller of the SMC is running in AArch32 mode
    or `PMF_SMC_GET_TIMESTAMP_64` when the caller is running in AArch64 mode.
x1: Timestamp identifier.
x2: The `mpidr` of the CPU for which the timestamp has to be retrieved.
    This can be the `mpidr` of a different core to the one initiating
    the SMC.  In that case, service specific cache maintenance may be
    required to ensure the updated copy of the timestamp is returned.
x3: A flags value that is either 0 or `PMF_CACHE_MAINT`.  If
    `PMF_CACHE_MAINT` is passed, then the PMF code will perform a
    cache invalidate before reading the timestamp.  This ensures
    an updated copy is returned.
```

The remaining arguments, `x4`, `cookie`, `handle` and `flags` are unused in this implementation.

PMF code structure

1. `pmf_main.c` consists of core functions that implement service registration, initialization, storing, dumping and retrieving timestamps.
2. `pmf_smc.c` contains the SMC handling for registered PMF services.
3. `pmf.h` contains the public interface to Performance Measurement Framework.
4. `pmf_asm_macros.S` consists of macros to facilitate capturing timestamps in assembly code.
5. `pmf_helpers.h` is an internal header used by `pmf.h`.

5.4.16 Armv8-A Architecture Extensions

TF-A makes use of Armv8-A Architecture Extensions where applicable. This section lists the usage of Architecture Extensions, and build flags controlling them.

In general, and unless individually mentioned, the build options `ARM_ARCH_MAJOR` and `ARM_ARCH_MINOR` select the Architecture Extension to target when building TF-A. Subsequent Arm Architecture Extensions are backward compatible with previous versions.

The build system only requires that `ARM_ARCH_MAJOR` and `ARM_ARCH_MINOR` have a valid numeric value. These build options only control whether or not Architecture Extension-specific code is included in the build. Otherwise, TF-A targets the base Armv8.0-A architecture; i.e. as if `ARM_ARCH_MAJOR` == 8 and `ARM_ARCH_MINOR` == 0, which are also their respective default values.

See also:

Build Options

For details on the Architecture Extension and available features, please refer to the respective Architecture Extension Supplement.

Armv8.1-A

This Architecture Extension is targeted when `ARM_ARCH_MAJOR` >= 8, or when `ARM_ARCH_MAJOR` == 8 and `ARM_ARCH_MINOR` >= 1.

- By default, a load-/store-exclusive instruction pair is used to implement spinlocks. The `USE_SPINLOCK_CAS` build option when set to 1 selects the spinlock implementation using the ARMv8.1-LSE Compare and Swap instruction. Notice this instruction is only available in AArch64 execution state, so the option is only available to AArch64 builds.

Armv8.2-A

- The presence of ARMv8.2-TTCNP is detected at runtime. When it is present, the Common not Private (TTBRn_ELx.CnP) bit is enabled to indicate that multiple Processing Elements in the same Inner Shareable domain use the same translation table entries for a given stage of translation for a particular translation regime.

Armv8.3-A

- Pointer authentication features of Armv8.3-A are unconditionally enabled in the Non-secure world so that lower ELs are allowed to use them without causing a trap to EL3.

  In order to enable the Secure world to use it, `CTX_INCLUDE_PAUTH_REGS` must be set to 1. This will add all pointer authentication system registers to the context that is saved when doing a world switch.

  The TF-A itself has support for pointer authentication at runtime that can be enabled by setting `BRANCH_PROTECTION` option to non-zero and `CTX_INCLUDE_PAUTH_REGS` to 1. This enables pointer authentication in BL1, BL2, BL31, and the TSP if it is used.

  These options are experimental features.

  Note that Pointer Authentication is enabled for Non-secure world irrespective of the value of these build flags if the CPU supports it.

  If `ARM_ARCH_MAJOR == 8` and `ARM_ARCH_MINOR >= 3` the code footprint of enabling PAuth is lower because the compiler will use the optimized PAuth instructions rather than the backwards-compatible ones.

Armv8.5-A

- Branch Target Identification feature is selected by `BRANCH_PROTECTION` option set to 1. This option defaults to 0 and this is an experimental feature.
- Memory Tagging Extension feature is unconditionally enabled for both worlds (at EL0 and S-EL0) if it is only supported at EL0. If instead it is implemented at all ELs, it is unconditionally enabled for only the normal world. To enable it for the secure world as well, the build option `CTX_INCLUDE_MTE_REGS` is required. If the hardware does not implement MTE support at all, it is always disabled, no matter what build options are used.

Armv7-A

This Architecture Extension is targeted when `ARM_ARCH_MAJOR` == 7.

There are several Armv7-A extensions available. Obviously the TrustZone extension is mandatory to support the TF-A bootloader and runtime services.

Platform implementing an Armv7-A system can to define from its target Cortex-A architecture through `ARM_CORTEX_A<X> = yes` in their `platform.mk` script. For example `ARM_CORTEX_A15=yes` for a Cortex-A15 target.

Platform can also set `ARM_WITH_NEON=yes` to enable neon support. Note that using neon at runtime has constraints on non secure world context. TF-A does not yet provide VFP context management.

Directive `ARM_CORTEX_A<x>` and `ARM_WITH_NEON` are used to set the toolchain target architecture directive.

Platform may choose to not define straight the toolchain target architecture directive by defining `MARCH32_DIRECTIVE`. I.e:

```
MARCH32_DIRECTIVE := -mach=armv7-a
```

5.4.17 Code Structure

TF-A code is logically divided between the three boot loader stages mentioned in the previous sections. The code is also divided into the following categories (present as directories in the source code):

- **Platform specific.** Choice of architecture specific code depends upon the platform.
- **Common code.** This is platform and architecture agnostic code.
- **Library code.** This code comprises of functionality commonly used by all other code. The PSCI implementation and other EL3 runtime frameworks reside as Library components.
- **Stage specific.** Code specific to a boot stage.
- **Drivers.**
- **Services.** EL3 runtime services (eg: SPD). Specific SPD services reside in the `services/spd` directory (e.g. `services/spd/tspd`).

Each boot loader stage uses code from one or more of the above mentioned categories. Based upon the above, the code layout looks like this:

```
Directory    Used by BL1?    Used by BL2?    Used by BL31?
bl1          Yes             No              No
bl2          No              Yes             No
bl31         No              No              Yes
plat         Yes             Yes             Yes
drivers      Yes             No              Yes
common       Yes             Yes             Yes
lib          Yes             Yes             Yes
services     No              No              Yes
```

The build system provides a non configurable build option IMAGE_BLx for each boot loader stage (where x = BL stage). e.g. for BL1 , IMAGE_BL1 will be defined by the build system. This enables TF-A to compile certain code only for specific boot loader stages

All assembler files have the `.S` extension. The linker source files for each boot stage have the extension `.ld.S`. These are processed by GCC to create the linker scripts which have the extension `.ld`.

FDTs provide a description of the hardware platform and are used by the Linux kernel at boot time. These can be found in the `fdts` directory.

References

- Trusted Board Boot Requirements CLIENT (TBBR-CLIENT) Armv8-A (ARM DEN0006D)
- Power State Coordination Interface PDD
- SMC Calling Convention
- *Interrupt Management Framework*

5.5 Interrupt Management Framework

This framework is responsible for managing interrupts routed to EL3. It also allows EL3 software to configure the interrupt routing behavior. Its main objective is to implement the following two requirements.

1. It should be possible to route interrupts meant to be handled by secure software (Secure interrupts) to EL3, when execution is in non-secure state (normal world). The framework should then take care of handing control of the interrupt to either software in EL3 or Secure-EL1 depending upon the software configuration and the GIC implementation. This requirement ensures that secure interrupts are under the control of the secure software with respect to their delivery and handling without the possibility of intervention from non-secure software.
2. It should be possible to route interrupts meant to be handled by non-secure software (Non-secure interrupts) to the last executed exception level in the normal world when the execution is in secure world at exception levels lower than EL3. This could be done with or without the knowledge of software executing in Secure-EL1/Secure-EL0. The choice of approach should be governed by the secure software. This requirement ensures that non-secure software is able to execute in tandem with the secure software without overriding it.

5.5.1 Concepts

Interrupt types

The framework categorises an interrupt to be one of the following depending upon the exception level(s) it is handled in.

1. Secure EL1 interrupt. This type of interrupt can be routed to EL3 or Secure-EL1 depending upon the security state of the current execution context. It is always handled in Secure-EL1.
2. Non-secure interrupt. This type of interrupt can be routed to EL3, Secure-EL1, Non-secure EL1 or EL2 depending upon the security state of the current execution context. It is always handled in either Non-secure EL1 or EL2.
3. EL3 interrupt. This type of interrupt can be routed to EL3 or Secure-EL1 depending upon the security state of the current execution context. It is always handled in EL3.

The following constants define the various interrupt types in the framework implementation.

```
#define INTR_TYPE_S_EL1       0
#define INTR_TYPE_EL3         1
#define INTR_TYPE_NS          2
```

Routing model

A type of interrupt can be either generated as an FIQ or an IRQ. The target exception level of an interrupt type is configured through the FIQ and IRQ bits in the Secure Configuration Register at EL3 (`SCR_EL3.FIQ` and `SCR_EL3.IRQ` bits). When `SCR_EL3.FIQ`=1, FIQs are routed to EL3. Otherwise they are routed to the First Exception Level (FEL) capable of handling interrupts. When `SCR_EL3.IRQ`=1, IRQs are routed to EL3. Otherwise they are routed to the FEL. This register is configured independently by EL3 software for each security state prior to entry into a lower exception level in that security state.

A routing model for a type of interrupt (generated as FIQ or IRQ) is defined as its target exception level for each security state. It is represented by a single bit for each security state. A value of `0` means that the interrupt should be routed to the FEL. A value of `1` means that the interrupt should be routed to EL3. A routing model is applicable only when execution is not in EL3.

The default routing model for an interrupt type is to route it to the FEL in either security state.

Valid routing models

The framework considers certain routing models for each type of interrupt to be incorrect as they conflict with the requirements mentioned in Section 1. The following sub-sections describe all the possible routing models and specify which ones are valid or invalid. EL3 interrupts are currently supported only for GIC version 3.0 (Arm GICv3) and only the Secure-EL1 and Non-secure interrupt types are supported for GIC version 2.0 (Arm GICv2) (see *Assumptions in Interrupt Management Framework*). The terminology used in the following sub-sections is explained below.

1. **CSS**. Current Security State. `0` when secure and `1` when non-secure
2. **TEL3**. Target Exception Level 3. `0` when targeted to the FEL. `1` when targeted to EL3.

Secure-EL1 interrupts

1. **CSS=0, TEL3=0**. Interrupt is routed to the FEL when execution is in secure state. This is a valid routing model as secure software is in control of handling secure interrupts.
2. **CSS=0, TEL3=1**. Interrupt is routed to EL3 when execution is in secure state. This is a valid routing model as secure software in EL3 can handover the interrupt to Secure-EL1 for handling.
3. **CSS=1, TEL3=0**. Interrupt is routed to the FEL when execution is in non-secure state. This is an invalid routing model as a secure interrupt is not visible to the secure software which violates the motivation behind the Arm Security Extensions.
4. **CSS=1, TEL3=1**. Interrupt is routed to EL3 when execution is in non-secure state. This is a valid routing model as secure software in EL3 can handover the interrupt to Secure-EL1 for handling.

Non-secure interrupts

1. **CSS=0, TEL3=0**. Interrupt is routed to the FEL when execution is in secure state. This allows the secure software to trap non-secure interrupts, perform its book-keeping and hand the interrupt to the non-secure software through EL3. This is a valid routing model as secure software is in control of how its execution is preempted by non-secure interrupts.
2. **CSS=0, TEL3=1**. Interrupt is routed to EL3 when execution is in secure state. This is a valid routing model as secure software in EL3 can save the state of software in Secure-EL1/Secure-EL0 before handing the interrupt to non-secure software. This model requires additional coordination between Secure-EL1 and EL3 software to ensure that the former's state is correctly saved by the latter.

3. **CSS=1, TEL3=0**. Interrupt is routed to FEL when execution is in non-secure state. This is a valid routing model as a non-secure interrupt is handled by non-secure software.
4. **CSS=1, TEL3=1**. Interrupt is routed to EL3 when execution is in non-secure state. This is an invalid routing model as there is no valid reason to route the interrupt to EL3 software and then hand it back to non-secure software for handling.

EL3 interrupts

1. **CSS=0, TEL3=0**. Interrupt is routed to the FEL when execution is in Secure-EL1/Secure-EL0. This is a valid routing model as secure software in Secure-EL1/Secure-EL0 is in control of how its execution is preempted by EL3 interrupt and can handover the interrupt to EL3 for handling.

   However, when `EL3_EXCEPTION_HANDLING` is `1`, this routing model is invalid as EL3 interrupts are unconditionally routed to EL3, and EL3 interrupts will always preempt Secure EL1/EL0 execution. See *exception handling* documentation.
2. **CSS=0, TEL3=1**. Interrupt is routed to EL3 when execution is in Secure-EL1/Secure-EL0. This is a valid routing model as secure software in EL3 can handle the interrupt.
3. **CSS=1, TEL3=0**. Interrupt is routed to the FEL when execution is in non-secure state. This is an invalid routing model as a secure interrupt is not visible to the secure software which violates the motivation behind the Arm Security Extensions.
4. **CSS=1, TEL3=1**. Interrupt is routed to EL3 when execution is in non-secure state. This is a valid routing model as secure software in EL3 can handle the interrupt.

Mapping of interrupt type to signal

The framework is meant to work with any interrupt controller implemented by a platform. A interrupt controller could generate a type of interrupt as either an FIQ or IRQ signal to the CPU depending upon the current security state. The mapping between the type and signal is known only to the platform. The framework uses this information to determine whether the IRQ or the FIQ bit should be programmed in `SCR_EL3` while applying the routing model for a type of interrupt. The platform provides this information through the `plat_interrupt_type_to_line()` API (described in the *Porting Guide*). For example, on the FVP port when the platform uses an Arm GICv2 interrupt controller, Secure-EL1 interrupts are signaled through the FIQ signal while Non-secure interrupts are signaled through the IRQ signal. This applies when execution is in either security state.

Effect of mapping of several interrupt types to one signal

It should be noted that if more than one interrupt type maps to a single interrupt signal, and if any one of the interrupt type sets **TEL3=1** for a particular security state, then interrupt signal will be routed to EL3 when in that security state. This means that all the other interrupt types using the same interrupt signal will be forced to the same routing model. This should be borne in mind when choosing the routing model for an interrupt type.

For example, in Arm GICv3, when the execution context is Secure-EL1/ Secure-EL0, both the EL3 and the non secure interrupt types map to the FIQ signal. So if either one of the interrupt type sets the routing model so that **TEL3=1** when **CSS=0**, the FIQ bit in `SCR_EL3` will be programmed to route the FIQ signal to EL3 when executing in Secure-EL1/Secure-EL0, thereby effectively routing the other interrupt type also to EL3.

5.5.2 Assumptions in Interrupt Management Framework

The framework makes the following assumptions to simplify its implementation.

1. Although the framework has support for 2 types of secure interrupts (EL3 and Secure-EL1 interrupt), only interrupt controller architectures like Arm GICv3 has architectural support for EL3 interrupts in the form of Group 0 interrupts. In Arm GICv2, all secure interrupts are assumed to be handled in Secure-EL1. They can be delivered to Secure-EL1 via EL3 but they cannot be handled in EL3.
2. Interrupt exceptions (`PSTATE.I` and `F` bits) are masked during execution in EL3.
3. Interrupt management: the following sections describe how interrupts are managed by the interrupt handling framework. This entails:
    1. Providing an interface to allow registration of a handler and specification of the routing model for a type of interrupt.
    2. Implementing support to hand control of an interrupt type to its registered handler when the interrupt is generated.

Both aspects of interrupt management involve various components in the secure software stack spanning from EL3 to Secure-EL1. These components are described in the section *Software components*. The framework stores information associated with each type of interrupt in the following data structure.

```
typedef struct intr_type_desc {
        interrupt_type_handler_t handler;
        uint32_t flags;
        uint32_t scr_el3[2];
} intr_type_desc_t;
```

The `flags` field stores the routing model for the interrupt type in bits[1:0]. Bit[0] stores the routing model when execution is in the secure state. Bit[1] stores the routing model when execution is in the non-secure state. As mentioned in Section *Routing model*, a value of `0` implies that the interrupt should be targeted to the FEL. A value of `1` implies that it should be targeted to EL3. The remaining bits are reserved and SBZ. The helper macro `set_interrupt_rm_flag()` should be used to set the bits in the `flags` parameter.

The `scr_el3[2]` field also stores the routing model but as a mapping of the model in the `flags` field to the corresponding bit in the `SCR_EL3` for each security state.

The framework also depends upon the platform port to configure the interrupt controller to distinguish between secure and non-secure interrupts. The platform is expected to be aware of the secure devices present in the system and their associated interrupt numbers. It should configure the interrupt controller to enable the secure interrupts, ensure that their priority is always higher than the non-secure interrupts and target them to the primary CPU. It should also export the interface described in the *Porting Guide* to enable handling of interrupts.

In the remainder of this document, for the sake of simplicity a Arm GICv2 system is considered and it is assumed that the FIQ signal is used to generate Secure-EL1 interrupts and the IRQ signal is used to generate non-secure interrupts in either security state. EL3 interrupts are not considered.

5.5.3 Software components

Roles and responsibilities for interrupt management are sub-divided between the following components of software running in EL3 and Secure-EL1. Each component is briefly described below.

1. EL3 Runtime Firmware. This component is common to all ports of TF-A.
2. Secure Payload Dispatcher (SPD) service. This service interfaces with the Secure Payload (SP) software which runs in Secure-EL1/Secure-EL0 and is responsible for switching execution between secure and non-secure states. A switch is triggered by a Secure Monitor Call and it uses the APIs exported by the Context management library to implement this functionality. Switching execution between the two security states is a requirement for interrupt management as well. This results in a significant dependency on the SPD service. TF-A implements an example Test Secure Payload Dispatcher (TSPD) service.

   An SPD service plugs into the EL3 runtime firmware and could be common to some ports of TF-A.
3. Secure Payload (SP). On a production system, the Secure Payload corresponds to a Secure OS which runs in Secure-EL1/Secure-EL0. It interfaces with the SPD service to manage communication with non-secure software. TF-A implements an example secure payload called Test Secure Payload (TSP) which runs only in Secure-EL1.

   A Secure payload implementation could be common to some ports of TF-A, just like the SPD service.

5.5.4 Interrupt registration

This section describes in detail the role of each software component (see *Software components*) during the registration of a handler for an interrupt type.

EL3 runtime firmware

This component declares the following prototype for a handler of an interrupt type.

```
typedef uint64_t (*interrupt_type_handler_t)(uint32_t id,
                                             uint32_t flags,
                                             void *handle,
                                             void *cookie);
```

The `id` is parameter is reserved and could be used in the future for passing the interrupt id of the highest pending interrupt only if there is a foolproof way of determining the id. Currently it contains `INTR_ID_UNAVAILABLE`.

The `flags` parameter contains miscellaneous information as follows.

1. Security state, bit[0]. This bit indicates the security state of the lower exception level when the interrupt was generated. A value of `1` means that it was in the non-secure state. A value of `0` indicates that it was in the secure state. This bit can be used by the handler to ensure that interrupt was generated and routed as per the routing model specified during registration.
2. Reserved, bits[31:1]. The remaining bits are reserved for future use.

The `handle` parameter points to the `cpu_context` structure of the current CPU for the security state specified in the `flags` parameter.

Once the handler routine completes, execution will return to either the secure or non-secure state. The handler routine must return a pointer to `cpu_context` structure of the current CPU for the target security state. On AArch64, this return value is currently ignored by the caller as the appropriate `cpu_context` to be used is expected to be set by the handler via the context management library APIs. A portable interrupt handler implementation must set the target context both in the structure pointed to by the returned pointer and via the context management library APIs. The handler should treat all error conditions as critical errors and take appropriate action within its implementation e.g. use assertion failures.

The runtime firmware provides the following API for registering a handler for a particular type of interrupt. A Secure Payload Dispatcher service should use this API to register a handler for Secure-EL1 and optionally for non-secure interrupts. This API also requires the caller to specify the routing model for the type of interrupt.

```
int32_t register_interrupt_type_handler(uint32_t type,
                                        interrupt_type_handler handler,
                                        uint64_t flags);
```

The `type` parameter can be one of the three interrupt types listed above i.e. `INTR_TYPE_S_EL1`, `INTR_TYPE_NS` & `INTR_TYPE_EL3`. The `flags` parameter is as described in Section 2.

The function will return `0` upon a successful registration. It will return `-EALREADY` in case a handler for the interrupt type has already been registered. If the `type` is unrecognised or the `flags` or the `handler` are invalid it will return `-EINVAL`.

Interrupt routing is governed by the configuration of the `SCR_EL3.FIQ/IRQ` bits prior to entry into a lower exception level in either security state. The context management library maintains a copy of the `SCR_EL3` system register for each security state in the `cpu_context` structure of each CPU. It exports the following APIs to let EL3 Runtime Firmware program and retrieve the routing model for each security state for the current CPU. The value of `SCR_EL3` stored in the `cpu_context` is used by the `el3_exit()` function to program the `SCR_EL3` register prior to returning from the EL3 exception level.

```
uint32_t cm_get_scr_el3(uint32_t security_state);
void cm_write_scr_el3_bit(uint32_t security_state,
                          uint32_t bit_pos,
                          uint32_t value);
```

`cm_get_scr_el3()` returns the value of the `SCR_EL3` register for the specified security state of the current CPU. `cm_write_scr_el3_bit()` writes a `0` or `1` to the bit specified by `bit_pos`. `register_interrupt_type_handler()` invokes `set_routing_model()` API which programs the `SCR_EL3` according to the routing model using the `cm_get_scr_el3()` and `cm_write_scr_el3_bit()` APIs.

It is worth noting that in the current implementation of the framework, the EL3 runtime firmware is responsible for programming the routing model. The SPD is responsible for ensuring that the routing model has been adhered to upon receiving an interrupt.

Secure payload dispatcher

A SPD service is responsible for determining and maintaining the interrupt routing model supported by itself and the Secure Payload. It is also responsible for ferrying interrupts between secure and non-secure software depending upon the routing model. It could determine the routing model at build time or at runtime. It must use this information to register a handler for each interrupt type using the `register_interrupt_type_handler()` API in EL3 runtime firmware.

If the routing model is not known to the SPD service at build time, then it must be provided by the SP as the result of its initialisation. The SPD should program the routing model only after SP initialisation has completed e.g. in the SPD initialisation function pointed to by the `bl32_init` variable.

The SPD should determine the mechanism to pass control to the Secure Payload after receiving an interrupt from the EL3 runtime firmware. This information could either be provided to the SPD service at build time or by the SP at runtime.

Test secure payload dispatcher behavior

Note: Where this document discusses `TSP_NS_INTR_ASYNC_PREEMPT` as being `1`, the same results also apply when `EL3_EXCEPTION_HANDLING` is `1`.

The TSPD only handles Secure-EL1 interrupts and is provided with the following routing model at build time.

- Secure-EL1 interrupts are routed to EL3 when execution is in non-secure state and are routed to the FEL when execution is in the secure state i.e **CSS=0, TEL3=0** & **CSS=1, TEL3=1** for Secure-EL1 interrupts
- When the build flag `TSP_NS_INTR_ASYNC_PREEMPT` is zero, the default routing model is used for non-secure interrupts. They are routed to the FEL in either security state i.e **CSS=0, TEL3=0** & **CSS=1, TEL3=0** for Non-secure interrupts.
- When the build flag `TSP_NS_INTR_ASYNC_PREEMPT` is defined to 1, then the non secure interrupts are routed to EL3 when execution is in secure state i.e **CSS=0, TEL3=1** for non-secure interrupts. This effectively preempts Secure-EL1. The default routing model is used for non secure interrupts in non-secure state. i.e **CSS=1, TEL3=0**.

It performs the following actions in the `tspd_init()` function to fulfill the requirements mentioned earlier.

1. It passes control to the Test Secure Payload to perform its initialisation. The TSP provides the address of the vector table `tsp_vectors` in the SP which also includes the handler for Secure-EL1 interrupts in the `sel1_intr_entry` field. The TSPD passes control to the TSP at this address when it receives a Secure-EL1 interrupt.

   The handover agreement between the TSP and the TSPD requires that the TSPD masks all interrupts (`PSTATE.DAIF` bits) when it calls `tsp_sel1_intr_entry()`. The TSP has to preserve the callee saved general purpose, SP_EL1/Secure-EL0, LR, VFP and system registers. It can use `x0-x18` to enable its C runtime.

2. The TSPD implements a handler function for Secure-EL1 interrupts. This function is registered with the EL3 runtime firmware using the `register_interrupt_type_handler()` API as follows

   ```
   /* Forward declaration */
   interrupt_type_handler tspd_secure_el1_interrupt_handler;
   int32_t rc, flags = 0;
   set_interrupt_rm_flag(flags, NON_SECURE);
   rc = register_interrupt_type_handler(INTR_TYPE_S_EL1,
                                    tspd_secure_el1_interrupt_handler,
                                    flags);
   if (rc)
       panic();
   ```

3. When the build flag `TSP_NS_INTR_ASYNC_PREEMPT` is defined to 1, the TSPD implements a handler function for non-secure interrupts. This function is registered with the EL3 runtime firmware using the `register_interrupt_type_handler()` API as follows

   ```
   /* Forward declaration */
   interrupt_type_handler tspd_ns_interrupt_handler;
   int32_t rc, flags = 0;
   set_interrupt_rm_flag(flags, SECURE);
   rc = register_interrupt_type_handler(INTR_TYPE_NS,
                                   tspd_ns_interrupt_handler,
                                   flags);
   if (rc)
       panic();
   ```

Secure payload

A Secure Payload must implement an interrupt handling framework at Secure-EL1 (Secure-EL1 IHF) to support its chosen interrupt routing model. Secure payload execution will alternate between the below cases.

1. In the code where IRQ, FIQ or both interrupts are enabled, if an interrupt type is targeted to the FEL, then it will be routed to the Secure-EL1 exception vector table. This is defined as the **asynchronous mode** of handling interrupts. This mode applies to both Secure-EL1 and non-secure interrupts.
2. In the code where both interrupts are disabled, if an interrupt type is targeted to the FEL, then execution will eventually migrate to the non-secure state. Any non-secure interrupts will be handled as described in the routing model where **CSS=1 and TEL3=0**. Secure-EL1 interrupts will be routed to EL3 (as per the routing model where **CSS=1 and TEL3=1**) where the SPD service will hand them to the SP. This is defined as the **synchronous mode** of handling interrupts.

The interrupt handling framework implemented by the SP should support one or both these interrupt handling models depending upon the chosen routing model.

The following list briefly describes how the choice of a valid routing model (see *Valid routing models*) effects the implementation of the Secure-EL1 IHF. If the choice of the interrupt routing model is not known to the SPD service at compile time, then the SP should pass this information to the SPD service at runtime during its initialisation phase.

As mentioned earlier, an Arm GICv2 system is considered and it is assumed that the FIQ signal is used to generate Secure-EL1 interrupts and the IRQ signal is used to generate non-secure interrupts in either security state.

Secure payload IHF design w.r.t secure-EL1 interrupts

1. **CSS=0, TEL3=0**. If `PSTATE.F=0`, Secure-EL1 interrupts will be triggered at one of the Secure-EL1 FIQ exception vectors. The Secure-EL1 IHF should implement support for handling FIQ interrupts asynchronously.

   If `PSTATE.F=1` then Secure-EL1 interrupts will be handled as per the synchronous interrupt handling model. The SP could implement this scenario by exporting a separate entrypoint for Secure-EL1 interrupts to the SPD service during the registration phase. The SPD service would also need to know the state of the system, general purpose and the `PSTATE` registers in which it should arrange to return execution to the SP. The SP should provide this information in an implementation defined way during the registration phase if it is not known to the SPD service at build time.
2. **CSS=1, TEL3=1**. Interrupts are routed to EL3 when execution is in non-secure state. They should be handled through the synchronous interrupt handling model as described in 1. above.
3. **CSS=0, TEL3=1**. Secure-EL1 interrupts are routed to EL3 when execution is in secure state. They will not be visible to the SP. The `PSTATE.F` bit in Secure-EL1/Secure-EL0 will not mask FIQs. The EL3 runtime firmware will call the handler registered by the SPD service for Secure-EL1 interrupts. Secure-EL1 IHF should then handle all Secure-EL1 interrupt through the synchronous interrupt handling model described in 1. above.

Secure payload IHF design w.r.t non-secure interrupts

1. **CSS=0, TEL3=0**. If `PSTATE.I=0`, non-secure interrupts will be triggered at one of the Secure-EL1 IRQ exception vectors . The Secure-EL1 IHF should co-ordinate with the SPD service to transfer execution to the non-secure state where the interrupt should be handled e.g the SP could allocate a function identifier to issue a SMC64 or SMC32 to the SPD service which indicates that the SP execution has been preempted by a non-secure interrupt. If this function identifier is not known to the SPD service at compile time then the SP could provide it during the registration phase.

   If `PSTATE.I=1` then the non-secure interrupt will pend until execution resumes in the non-secure state.

2. **CSS=0, TEL3=1**. Non-secure interrupts are routed to EL3. They will not be visible to the SP. The `PSTATE.I` bit in Secure-EL1/Secure-EL0 will have not effect. The SPD service should register a non-secure interrupt handler which should save the SP state correctly and resume execution in the non-secure state where the interrupt will be handled. The Secure-EL1 IHF does not need to take any action.
3. **CSS=1, TEL3=0**. Non-secure interrupts are handled in the FEL in non-secure state (EL1/EL2) and are not visible to the SP. This routing model does not affect the SP behavior.

A Secure Payload must also ensure that all Secure-EL1 interrupts are correctly configured at the interrupt controller by the platform port of the EL3 runtime firmware. It should configure any additional Secure-EL1 interrupts which the EL3 runtime firmware is not aware of through its platform port.

Test secure payload behavior

The routing model for Secure-EL1 and non-secure interrupts chosen by the TSP is described in Section *Secure Payload Dispatcher*. It is known to the TSPD service at build time.

The TSP implements an entrypoint (`tsp_sel1_intr_entry()`) for handling Secure-EL1 interrupts taken in non-secure state and routed through the TSPD service (synchronous handling model). It passes the reference to this entrypoint via `tsp_vectors` to the TSPD service.

The TSP also replaces the default exception vector table referenced through the `early_exceptions` variable, with a vector table capable of handling FIQ and IRQ exceptions taken at the same (Secure-EL1) exception level. This table is referenced through the `tsp_exceptions` variable and programmed into the VBAR_EL1. It caters for the asynchronous handling model.

The TSP also programs the Secure Physical Timer in the Arm Generic Timer block to raise a periodic interrupt (every half a second) for the purpose of testing interrupt management across all the software components listed in *Software components*.

5.5.5 Interrupt handling

This section describes in detail the role of each software component (see Section *Software components*) in handling an interrupt of a particular type.

EL3 runtime firmware

The EL3 runtime firmware populates the IRQ and FIQ exception vectors referenced by the `runtime_exceptions` variable as follows.

1. IRQ and FIQ exceptions taken from the current exception level with `SP_EL0` or `SP_EL3` are reported as irrecoverable error conditions. As mentioned earlier, EL3 runtime firmware always executes with the `PSTATE.I` and `PSTATE.F` bits set.
2. The following text describes how the IRQ and FIQ exceptions taken from a lower exception level using AArch64 or AArch32 are handled.

When an interrupt is generated, the vector for each interrupt type is responsible for:

1. Saving the entire general purpose register context (x0-x30) immediately upon exception entry. The registers are saved in the per-cpu `cpu_context` data structure referenced by the `SP_EL3`register.
2. Saving the `ELR_EL3`, `SP_EL0` and `SPSR_EL3` system registers in the per-cpu `cpu_context` data structure referenced by the `SP_EL3` register.
3. Switching to the C runtime stack by restoring the `CTX_RUNTIME_SP` value from the per-cpu `cpu_context` data structure in `SP_EL0` and executing the `msr spsel, #0` instruction.

4. Determining the type of interrupt. Secure-EL1 interrupts will be signaled at the FIQ vector. Non-secure interrupts will be signaled at the IRQ vector. The platform should implement the following API to determine the type of the pending interrupt.

   ```
   uint32_t plat_ic_get_interrupt_type(void);
   ```

   It should return either `INTR_TYPE_S_EL1` or `INTR_TYPE_NS`.

5. Determining the handler for the type of interrupt that has been generated. The following API has been added for this purpose.

   ```
   interrupt_type_handler get_interrupt_type_handler(uint32_t interrupt_type);
   ```

   It returns the reference to the registered handler for this interrupt type. The `handler` is retrieved from the `intr_type_desc_t` structure as described in Section 2. `NULL` is returned if no handler has been registered for this type of interrupt. This scenario is reported as an irrecoverable error condition.

6. Calling the registered handler function for the interrupt type generated. The `id` parameter is set to `INTR_ID_UNAVAILABLE` currently. The id along with the current security state and a reference to the `cpu_context_t` structure for the current security state are passed to the handler function as its arguments.

   The handler function returns a reference to the per-cpu `cpu_context_t` structure for the target security state.

7. Calling `el3_exit()` to return from EL3 into a lower exception level in the security state determined by the handler routine. The `el3_exit()` function is responsible for restoring the register context from the `cpu_context_t` data structure for the target security state.

Secure payload dispatcher

Interrupt entry

The SPD service begins handling an interrupt when the EL3 runtime firmware calls the handler function for that type of interrupt. The SPD service is responsible for the following:

1. Validating the interrupt. This involves ensuring that the interrupt was generated according to the interrupt routing model specified by the SPD service during registration. It should use the security state of the exception level (passed in the `flags` parameter of the handler) where the interrupt was taken from to determine this. If the interrupt is not recognised then the handler should treat it as an irrecoverable error condition.

   An SPD service can register a handler for Secure-EL1 and/or Non-secure interrupts. A non-secure interrupt should never be routed to EL3 from from non-secure state. Also if a routing model is chosen where Secure-EL1 interrupts are routed to S-EL1 when execution is in Secure state, then a S-EL1 interrupt should never be routed to EL3 from secure state. The handler could use the security state flag to check this.

2. Determining whether a context switch is required. This depends upon the routing model and interrupt type. For non secure and S-EL1 interrupt, if the security state of the execution context where the interrupt was generated is not the same as the security state required for handling the interrupt, a context switch is required. The following 2 cases require a context switch from secure to non-secure or vice-versa:

   1. A Secure-EL1 interrupt taken from the non-secure state should be routed to the Secure Payload.
   2. A non-secure interrupt taken from the secure state should be routed to the last known non-secure exception level.

   The SPD service must save the system register context of the current security state. It must then restore the system register context of the target security state. It should use the `cm_set_next_eret_context()` API to ensure that the next `cpu_context` to be restored is of the target security state.

If the target state is secure then execution should be handed to the SP as per the synchronous interrupt handling model it implements. A Secure-EL1 interrupt can be routed to EL3 while execution is in the SP. This implies that SP execution can be preempted while handling an interrupt by a another higher priority Secure-EL1 interrupt or a EL3 interrupt. The SPD service should be able to handle this preemption or manage secure interrupt priorities before handing control to the SP.

3. Setting the return value of the handler to the per-cpu `cpu_context` if the interrupt has been successfully validated and ready to be handled at a lower exception level.

The routing model allows non-secure interrupts to interrupt Secure-EL1 when in secure state if it has been configured to do so. The SPD service and the SP should implement a mechanism for routing these interrupts to the last known exception level in the non-secure state. The former should save the SP context, restore the non-secure context and arrange for entry into the non-secure state so that the interrupt can be handled.

Interrupt exit

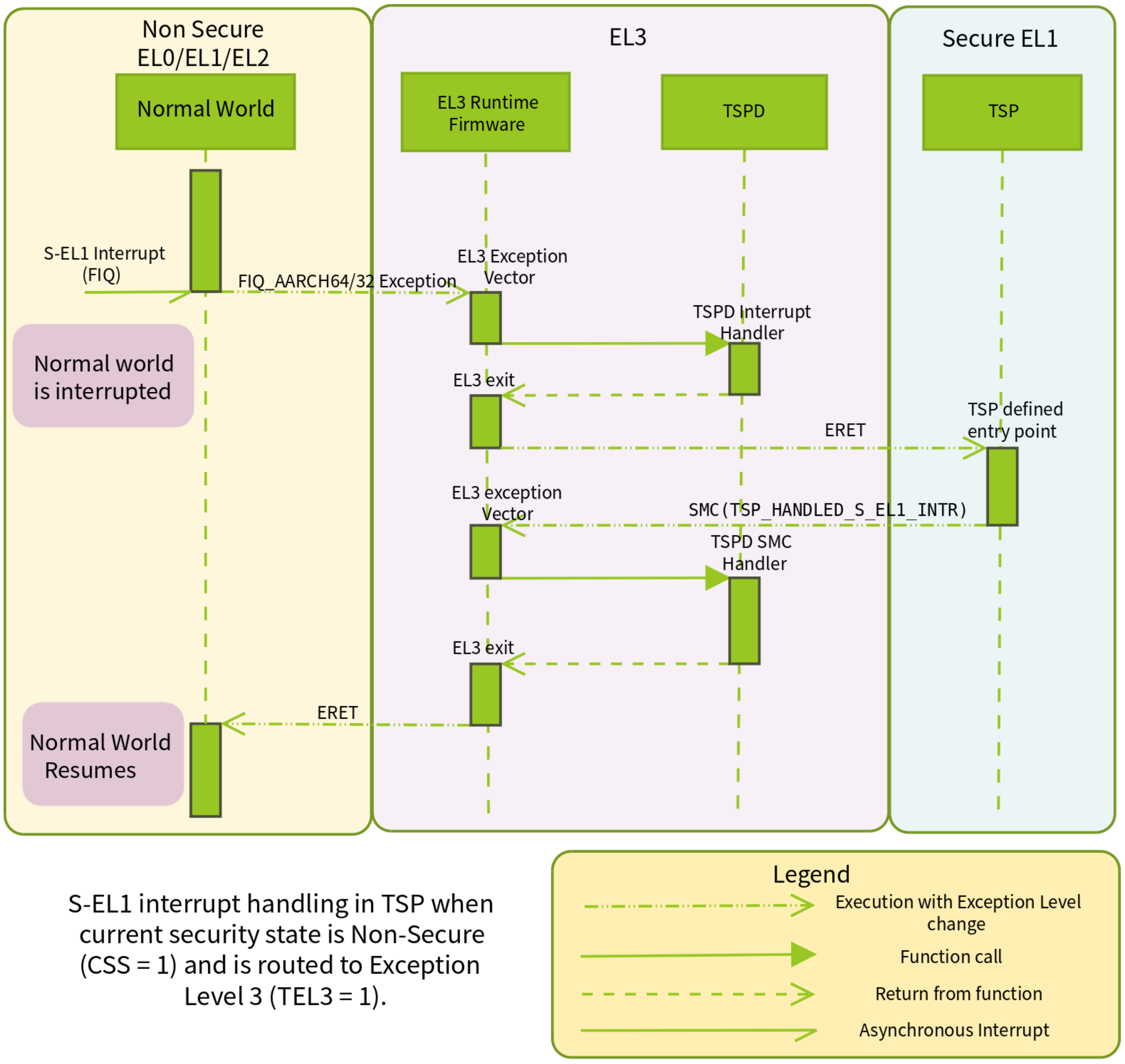
When the Secure Payload has finished handling a Secure-EL1 interrupt, it could return control back to the SPD service through a SMC32 or SMC64. The SPD service should handle this secure monitor call so that execution resumes in the exception level and the security state from where the Secure-EL1 interrupt was originally taken.

Test secure payload dispatcher Secure-EL1 interrupt handling

The example TSPD service registers a handler for Secure-EL1 interrupts taken from the non-secure state. During execution in S-EL1, the TSPD expects that the Secure-EL1 interrupts are handled in S-EL1 by TSP. Its handler `tspd_secure_el1_interrupt_handler()` expects only to be invoked for Secure-EL1 originating from the non-secure state. It takes the following actions upon being invoked.

1. It uses the security state provided in the `flags` parameter to ensure that the secure interrupt originated from the non-secure state. It asserts if this is not the case.
2. It saves the system register context for the non-secure state by calling `cm_el1_sysregs_context_save(NON_SECURE);`.
3. It sets the `ELR_EL3` system register to `tsp_sel1_intr_entry` and sets the `SPSR_EL3.DAIF` bits in the secure CPU context. It sets `x0` to `TSP_HANDLE_SEL1_INTR_AND_RETURN`. If the TSP was preempted earlier by a non secure interrupt during `yielding` SMC processing, save the registers that will be trashed, which is the `ELR_EL3` and `SPSR_EL3`, in order to be able to re-enter TSP for Secure-EL1 interrupt processing. It does not need to save any other secure context since the TSP is expected to preserve it (see section *Test secure payload dispatcher behavior*).
4. It restores the system register context for the secure state by calling `cm_el1_sysregs_context_restore(SECURE);`.
5. It ensures that the secure CPU context is used to program the next exception return from EL3 by calling `cm_set_next_eret_context(SECURE);`.
6. It returns the per-cpu `cpu_context` to indicate that the interrupt can now be handled by the SP. `x1` is written with the value of `elr_el3` register for the non-secure state. This information is used by the SP for debugging purposes.

The figure below describes how the interrupt handling is implemented by the TSPD when a Secure-EL1 interrupt is generated when execution is in the non-secure state.



S-EL1 interrupt handling in TSP when current security state is Non-Secure (CSS = 1) and is routed to Exception Level 3 (TEL3 = 1).

The TSP issues an SMC with `TSP_HANDLED_S_EL1_INTR` as the function identifier to signal completion of interrupt handling.

The TSPD service takes the following actions in `tspd_smc_handler()` function upon receiving an SMC with `TSP_HANDLED_S_EL1_INTR` as the function identifier:

1. It ensures that the call originated from the secure state otherwise execution returns to the non-secure state with `SMC_UNK` in `x0`.
2. It restores the saved `ELR_EL3` and `SPSR_EL3` system registers back to the secure CPU context (see step 3 above) in case the TSP had been preempted by a non secure interrupt earlier.
3. It restores the system register context for the non-secure state by calling `cm_el1_sysregs_context_restore(NON_SECURE)`.
4. It ensures that the non-secure CPU context is used to program the next exception return from EL3 by calling `cm_set_next_eret_context(NON_SECURE)`.
5. `tspd_smc_handler()` returns a reference to the non-secure `cpu_context` as the return value.

Test secure payload dispatcher non-secure interrupt handling

The TSP in Secure-EL1 can be preempted by a non-secure interrupt during `yielding` SMC processing or by a higher priority EL3 interrupt during Secure-EL1 interrupt processing. When `EL3_EXCEPTION_HANDLING` is `0`, only non-secure interrupts can cause preemption of TSP since there are no EL3 interrupts in the system. With `EL3_EXCEPTION_HANDLING=1` however, any EL3 interrupt may preempt Secure execution.

It should be noted that while TSP is preempted, the TSPD only allows entry into the TSP either for Secure-EL1 interrupt handling or for resuming the preempted `yielding` SMC in response to the `TSP_FID_RESUME` SMC from the normal world. (See Section *Implication of preempted SMC on Non-Secure Software*).

The non-secure interrupt triggered in Secure-EL1 during `yielding` SMC processing can be routed to either EL3 or Secure-EL1 and is controlled by build option `TSP_NS_INTR_ASYNC_PREEMPT` (see Section *Test secure payload dispatcher behavior*). If the build option is set, the TSPD will set the routing model for the non-secure interrupt to be routed to EL3 from secure state i.e. **TEL3=1, CSS=0** and registers `tspd_ns_interrupt_handler()` as the non-secure interrupt handler. The `tspd_ns_interrupt_handler()` on being invoked ensures that the interrupt originated from the secure state and disables routing of non-secure interrupts from secure state to EL3. This is to prevent further preemption (by a non-secure interrupt) when TSP is reentered for handling Secure-EL1 interrupts that triggered while execution was in the normal world. The `tspd_ns_interrupt_handler()` then invokes `tspd_handle_sp_preemption()` for further handling.

If the `TSP_NS_INTR_ASYNC_PREEMPT` build option is zero (default), the default routing model for non-secure interrupt in secure state is in effect i.e. **TEL3=0, CSS=0**. During `yielding` SMC processing, the IRQ exceptions are unmasked i.e. `PSTATE.I=0`, and a non-secure interrupt will trigger at Secure-EL1 IRQ exception vector. The TSP saves the general purpose register context and issues an SMC with `TSP_PREEMPTED` as the function identifier to signal preemption of TSP. The TSPD SMC handler, `tspd_smc_handler()`, ensures that the SMC call originated from the secure state otherwise execution returns to the non-secure state with `SMC_UNK` in `x0`. It then invokes `tspd_handle_sp_preemption()` for further handling.

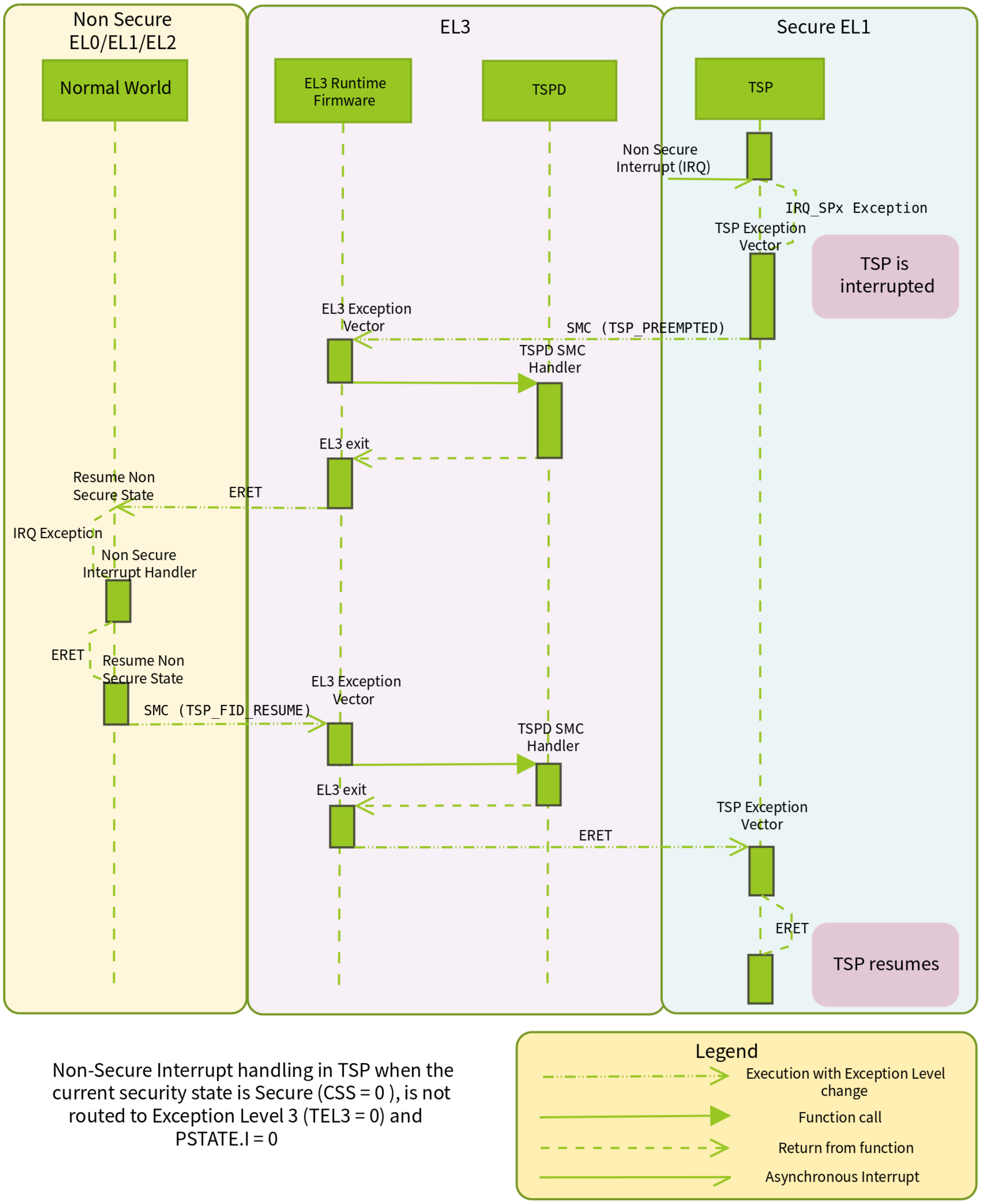
The `tspd_handle_sp_preemption()` takes the following actions upon being invoked:

1. It saves the system register context for the secure state by calling `cm_el1_sysregs_context_save(SECURE)`.
2. It restores the system register context for the non-secure state by calling `cm_el1_sysregs_context_restore(NON_SECURE)`.
3. It ensures that the non-secure CPU context is used to program the next exception return from EL3 by calling `cm_set_next_eret_context(NON_SECURE)`.
4. `SMC_PREEMPTED` is set in x0 and return to non secure state after restoring non secure context.

The Normal World is expected to resume the TSP after the `yielding` SMC preemption by issuing an SMC with `TSP_FID_RESUME` as the function identifier (see section *Implication of preempted SMC on Non-Secure Software*). The TSPD service takes the following actions in `tspd_smc_handler()` function upon receiving this SMC:

1. It ensures that the call originated from the non secure state. An assertion is raised otherwise.
2. Checks whether the TSP needs a resume i.e check if it was preempted. It then saves the system register context for the non-secure state by calling `cm_el1_sysregs_context_save(NON_SECURE)`.
3. Restores the secure context by calling `cm_el1_sysregs_context_restore(SECURE)`
4. It ensures that the secure CPU context is used to program the next exception return from EL3 by calling `cm_set_next_eret_context(SECURE)`.
5. `tspd_smc_handler()` returns a reference to the secure `cpu_context` as the return value.

The figure below describes how the TSP/TSPD handle a non-secure interrupt when it is generated during execution in the TSP with `PSTATE.I` = 0 when the `TSP_NS_INTR_ASYNC_PREEMPT` build flag is 0.



Non-Secure Interrupt handling in TSP when the current security state is Secure (CSS = 0), is not routed to Exception Level 3 (TEL3 = 0) and PSTATE.I = 0

Secure payload interrupt handling

The SP should implement one or both of the synchronous and asynchronous interrupt handling models depending upon the interrupt routing model it has chosen (as described in section *Secure Payload*).

In the synchronous model, it should begin handling a Secure-EL1 interrupt after receiving control from the SPD service at an entrypoint agreed upon during build time or during the registration phase. Before handling the interrupt, the SP should save any Secure-EL1 system register context which is needed for resuming normal execution in the SP later e.g. `SPSR_EL1`, `ELR_EL1`. After handling the interrupt, the SP could return control back to the exception level and security state where the interrupt was originally taken from. The SP should use an SMC32 or SMC64 to ask the SPD service to do this.

In the asynchronous model, the Secure Payload is responsible for handling non-secure and Secure-EL1 interrupts at the IRQ and FIQ vectors in its exception vector table when `PSTATE.I` and `PSTATE.F` bits are 0. As described earlier, when a non-secure interrupt is generated, the SP should coordinate with the SPD service to pass control back to the non-secure state in the last known exception level. This will allow the non-secure interrupt to be handled in the non-secure state.

Test secure payload behavior

The TSPD hands control of a Secure-EL1 interrupt to the TSP at the `tsp_sel1_intr_entry()`. The TSP handles the interrupt while ensuring that the handover agreement described in Section *Test secure payload dispatcher behavior* is maintained. It updates some statistics by calling `tsp_update_sync_sel1_intr_stats()`. It then calls `tsp_common_int_handler()` which.

1. Checks whether the interrupt is the secure physical timer interrupt. It uses the platform API `plat_ic_get_pending_interrupt_id()` to get the interrupt number. If it is not the secure physical timer interrupt, then that means that a higher priority interrupt has preempted it. Invoke `tsp_handle_preemption()` to handover control back to EL3 by issuing an SMC with `TSP_PREEMPTED` as the function identifier.
2. Handles the secure timer interrupt interrupt by acknowledging it using the `plat_ic_acknowledge_interrupt()` platform API, calling `tsp_generic_timer_handler()` to reprogram the secure physical generic timer and calling the `plat_ic_end_of_interrupt()` platform API to signal end of interrupt processing.

The TSP passes control back to the TSPD by issuing an SMC64 with `TSP_HANDLED_S_EL1_INTR` as the function identifier.

The TSP handles interrupts under the asynchronous model as follows.

1. Secure-EL1 interrupts are handled by calling the `tsp_common_int_handler()` function. The function has been described above.
2. Non-secure interrupts are handled by calling the `tsp_common_int_handler()` function which ends up invoking `tsp_handle_preemption()` and issuing an SMC64 with `TSP_PREEMPTED` as the function identifier. Execution resumes at the instruction that follows this SMC instruction when the TSPD hands control to the TSP in response to an SMC with `TSP_FID_RESUME` as the function identifier from the non-secure state (see section *Test secure payload dispatcher non-secure interrupt handling*).

5.5.6 Other considerations

Implication of preempted SMC on Non-Secure Software

A `yielding` SMC call to Secure payload can be preempted by a non-secure interrupt and the execution can return to the non-secure world for handling the interrupt (For details on `yielding` SMC refer SMC calling convention). In this case, the SMC call has not completed its execution and the execution must return back to the secure payload to resume the preempted SMC call. This can be achieved by issuing an SMC call which instructs to resume the preempted SMC.

A `fast` SMC cannot be preempted and hence this case will not happen for a fast SMC call.

In the Test Secure Payload implementation, `TSP_FID_RESUME` is designated as the resume SMC FID. It is important to note that `TSP_FID_RESUME` is a `yielding` SMC which means it too can be be preempted. The typical non secure software sequence for issuing a `yielding` SMC would look like this, assuming `P.STATE.I=0` in the non secure state :

```
int rc;
rc = smc(TSP_YIELD_SMC_FID, ...);     /* Issue a Yielding SMC call */
/* The pending non-secure interrupt is handled by the interrupt handler
   and returns back here. */
while (rc == SMC_PREEMPTED) {        /* Check if the SMC call is preempted */
    rc = smc(TSP_FID_RESUME);        /* Issue resume SMC call */
}
```

The `TSP_YIELD_SMC_FID` is any `yielding` SMC function identifier and the smc() function invokes a SMC call with the required arguments. The pending non-secure interrupt causes an IRQ exception and the IRQ handler registered at the exception vector handles the non-secure interrupt and returns. The return value from the SMC call is tested for `SMC_PREEMPTED` to check whether it is preempted. If it is, then the resume SMC call `TSP_FID_RESUME` is issued. The return value of the SMC call is tested again to check if it is preempted. This is done in a loop till the SMC call succeeds or fails. If a `yielding` SMC is preempted, until it is resumed using `TSP_FID_RESUME` SMC and completed, the current TSPD prevents any other SMC call from re-entering TSP by returning `SMC_UNK` error.

5.6 PSCI Power Domain Tree Structure

5.6.1 Requirements

1. A platform must export the `plat_get_aff_count()` and `plat_get_aff_state()` APIs to enable the generic PSCI code to populate a tree that describes the hierarchy of power domains in the system. This approach is inflexible because a change to the topology requires a change in the code.

   It would be much simpler for the platform to describe its power domain tree in a data structure.

2. The generic PSCI code generates MPIDRs in order to populate the power domain tree. It also uses an MPIDR to find a node in the tree. The assumption that a platform will use exactly the same MPIDRs as generated by the generic PSCI code is not scalable. The use of an MPIDR also restricts the number of levels in the power domain tree to four.

   Therefore, there is a need to decouple allocation of MPIDRs from the mechanism used to populate the power domain topology tree.

3. The current arrangement of the power domain tree requires a binary search over the sibling nodes at a particular level to find a specified power domain node. During a power management operation, the tree is traversed from a 'start' to an 'end' power level. The binary search is required to find the node at each level. The natural way to perform this traversal is to start from a leaf node and follow the parent node pointer to reach the end level.

   Therefore, there is a need to define data structures that implement the tree in a way which facilitates such a traversal.

4. The attributes of a core power domain differ from the attributes of power domains at higher levels. For example, only a core power domain can be identified using an MPIDR. There is no requirement to perform state coordination while performing a power management operation on the core power domain.

   Therefore, there is a need to implement the tree in a way which facilitates this distinction between a leaf and non-leaf node and any associated optimizations.

5.6.2 Design

Describing a power domain tree

To fulfill requirement 1., the existing platform APIs `plat_get_aff_count()` and `plat_get_aff_state()` have been removed. A platform must define an array of unsigned chars such that:

1. The first entry in the array specifies the number of power domains at the highest power level implemented in the platform. This caters for platforms where the power domain tree does not have a single root node, for example, the FVP has two cluster power domains at the highest level (1).
2. Each subsequent entry corresponds to a power domain and contains the number of power domains that are its direct children.
3. The size of the array minus the first entry will be equal to the number of non-leaf power domains.
4. The value in each entry in the array is used to find the number of entries to consider at the next level. The sum of the values (number of children) of all the entries at a level specifies the number of entries in the array for the next level.

The following example power domain topology tree will be used to describe the above text further. The leaf and non-leaf nodes in this tree have been numbered separately.

```
                               +-+
                               |0|
                               +-+
                              /    \
                             /      \
                            /        \
                           /          \
                          /            \
                         /              \
                        /                \
                       /                  \
                      /                    \
                     /                      \
                  +-+                       +-+
                  |1|                       |2|
                  +-+                       +-+
                 /   \                     /   \
                /     \                   /     \
```

(continues on next page)

(continued from previous page)

```
               /       \                 /       \
              /         \               /         \
           +-+           +-+         +-+           +-+
           |3|           |4|         |5|           |6|
           +-+           +-+         +-+           +-+
  +---+-----+    +----+----|     +----+----+     +----+-----+------+
  |   |     |    |    |    |     |    |    |     |    |     |      |
  |   |     |    |    |    |     |    |    |     |    |     |      |
  v   v     v    v    v    v     v    v    v     v    v     v      v
+-+  +-+   +-+  +-+  +-+  +-+   +-+  +-+  +-+   +-+  +--+  +--+  +--+
|0|  |1|   |2|  |3|  |4|  |5|   |6|  |7|  |8|   |9|  |10|  |11|  |12|
+-+  +-+   +-+  +-+  +-+  +-+   +-+  +-+  +-+   +-+  +--+  +--+  +--+
```

This tree is defined by the platform as the array described above as follows:

```
#define PLAT_NUM_POWER_DOMAINS       20
#define PLATFORM_CORE_COUNT          13
#define PSCI_NUM_NON_CPU_PWR_DOMAINS \
                   (PLAT_NUM_POWER_DOMAINS - PLATFORM_CORE_COUNT)

unsigned char plat_power_domain_tree_desc[] = { 1, 2, 2, 2, 3, 3, 3, 4};
```

Removing assumptions about MPIDRs used in a platform

To fulfill requirement 2., it is assumed that the platform assigns a unique number (core index) between `0` and `PLAT_CORE_COUNT - 1` to each core power domain. MPIDRs could be allocated in any manner and will not be used to populate the tree.

`plat_core_pos_by_mpidr(mpidr)` will return the core index for the core corresponding to the MPIDR. It will return an error (-1) if an MPIDR is passed which is not allocated or corresponds to an absent core. The semantics of this platform API have changed since it is required to validate the passed MPIDR. It has been made a mandatory API as a result.

Another mandatory API, `plat_my_core_pos()` has been added to return the core index for the calling core. This API provides a more lightweight mechanism to get the index since there is no need to validate the MPIDR of the calling core.

The platform should assign the core indices (as illustrated in the diagram above) such that, if the core nodes are numbered from left to right, then the index for a core domain will be the same as the index returned by `plat_core_pos_by_mpidr()` or `plat_my_core_pos()` for that core. This relationship allows the core nodes to be allocated in a separate array (requirement 4.) during `psci_setup()` in such an order that the index of the core in the array is the same as the return value from these APIs.

Dealing with holes in MPIDR allocation

For platforms where the number of allocated MPIDRs is equal to the number of core power domains, for example, Juno and FVPs, the logic to convert an MPIDR to a core index should remain unchanged. Both Juno and FVP use a simple collision proof hash function to do this.

It is possible that on some platforms, the allocation of MPIDRs is not contiguous or certain cores have been disabled. This essentially means that the MPIDRs have been sparsely allocated, that is, the size of the range of MPIDRs used by the platform is not equal to the number of core power domains.

The platform could adopt one of the following approaches to deal with this scenario:

1. Implement more complex logic to convert a valid MPIDR to a core index while maintaining the relationship described earlier. This means that the power domain tree descriptor will not describe any core power domains which are disabled or absent. Entries will not be allocated in the tree for these domains.
2. Treat unallocated MPIDRs and disabled cores as absent but still describe them in the power domain descriptor, that is, the number of core nodes described is equal to the size of the range of MPIDRs allocated. This approach will lead to memory wastage since entries will be allocated in the tree but will allow use of a simpler logic to convert an MPIDR to a core index.

Traversing through and distinguishing between core and non-core power domains

To fulfill requirement 3 and 4, separate data structures have been defined to represent leaf and non-leaf power domain nodes in the tree.

```
/*******************************************************************************
 * The following two data structures implement the power domain tree. The tree
 * is used to track the state of all the nodes i.e. power domain instances
 * described by the platform. The tree consists of nodes that describe CPU power
 * domains i.e. leaf nodes and all other power domains which are parents of a
 * CPU power domain i.e. non-leaf nodes.
 ******************************************************************************/
typedef struct non_cpu_pwr_domain_node {
    /*
     * Index of the first CPU power domain node level 0 which has this node
     * as its parent.
     */
    unsigned int cpu_start_idx;

    /*
     * Number of CPU power domains which are siblings of the domain indexed
     * by 'cpu_start_idx' i.e. all the domains in the range 'cpu_start_idx
     * -> cpu_start_idx + ncpus' have this node as their parent.
     */
    unsigned int ncpus;

    /* Index of the parent power domain node */
    unsigned int parent_node;

    -----
} non_cpu_pd_node_t;

typedef struct cpu_pwr_domain_node {
    u_register_t mpidr;

    /* Index of the parent power domain node */
    unsigned int parent_node;

    -----
} cpu_pd_node_t;
```

The power domain tree is implemented as a combination of the following data structures.

```
non_cpu_pd_node_t psci_non_cpu_pd_nodes[PSCI_NUM_NON_CPU_PWR_DOMAINS];
cpu_pd_node_t psci_cpu_pd_nodes[PLATFORM_CORE_COUNT];
```

Populating the power domain tree

The populate_power_domain_tree() function in psci_setup.c implements the algorithm to parse the power domain descriptor exported by the platform to populate the two arrays. It is essentially a breadth-first-search. The nodes for each level starting from the root are laid out one after another in the psci_non_cpu_pd_nodes and psci_cpu_pd_nodes arrays as follows:

```
psci_non_cpu_pd_nodes -> [[Level 3 nodes][Level 2 nodes][Level 1 nodes]]
psci_cpu_pd_nodes -> [Level 0 nodes]
```

For the example power domain tree illustrated above, the psci_cpu_pd_nodes will be populated as follows. The value in each entry is the index of the parent node. Other fields have been ignored for simplicity.

```
      +-------------+     ^
CPU0  |      3      |     |
      +-------------+     |
CPU1  |      3      |     |
      +-------------+     |
CPU2  |      3      |     |
      +-------------+     |
CPU3  |      4      |     |
      +-------------+     |
CPU4  |      4      |     |
      +-------------+     |
CPU5  |      4      |     | PLATFORM_CORE_COUNT
      +-------------+     |
CPU6  |      5      |     |
      +-------------+     |
CPU7  |      5      |     |
      +-------------+     |
CPU8  |      5      |     |
      +-------------+     |
CPU9  |      6      |     |
      +-------------+     |
CPU10 |      6      |     |
      +-------------+     |
CPU11 |      6      |     |
      +-------------+     |
CPU12 |      6      |     v
      +-------------+
```

The psci_non_cpu_pd_nodes array will be populated as follows. The value in each entry is the index of the parent node.

```
      +-------------+     ^
PD0   |     -1      |     |
      +-------------+     |
PD1   |      0      |     |
      +-------------+     |
PD2   |      0      |     |
      +-------------+     |
PD3   |      1      |     | PLAT_NUM_POWER_DOMAINS -
      +-------------+     | PLATFORM_CORE_COUNT
PD4   |      1      |     |
      +-------------+     |
PD5   |      2      |     |
      +-------------+     |
```

(continues on next page)

(continued from previous page)

```
PD6   |       2      |     |
      +-------------+      v
```

Each core can find its node in the `psci_cpu_pd_nodes` array using the `plat_my_core_pos()` function. When a core is turned on, the normal world provides an MPIDR. The `plat_core_pos_by_mpidr()` function is used to validate the MPIDR before using it to find the corresponding core node. The non-core power domain nodes do not need to be identified.

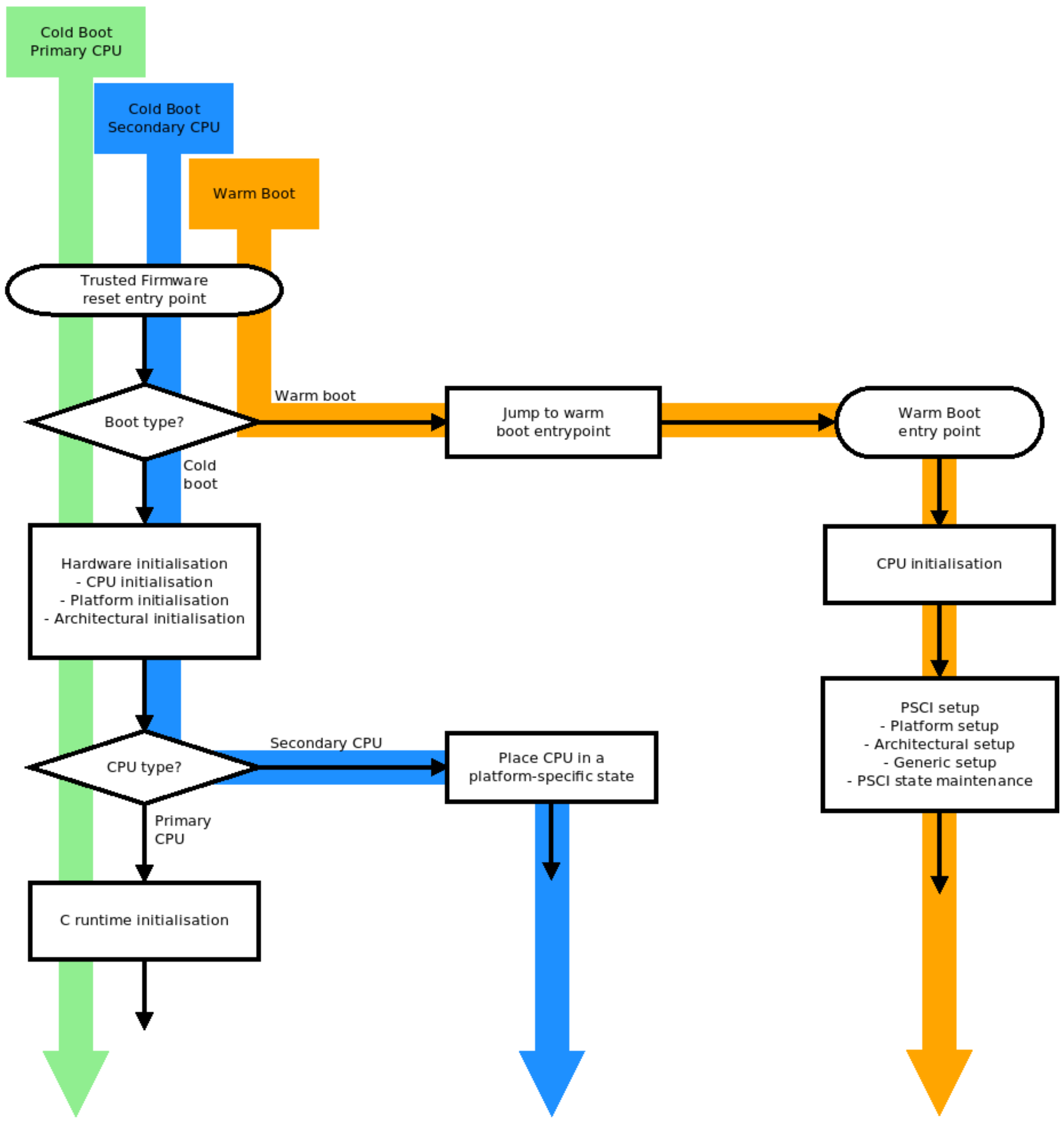
5.7 CPU Reset

This document describes the high-level design of the framework to handle CPU resets in Trusted Firmware-A (TF-A). It also describes how the platform integrator can tailor this code to the system configuration to some extent, resulting in a simplified and more optimised boot flow.

This document should be used in conjunction with the *Firmware Design* document which provides greater implementation details around the reset code, specifically for the cold boot path.

5.7.1 General reset code flow

The TF-A reset code is implemented in BL1 by default. The following high-level diagram illustrates this:



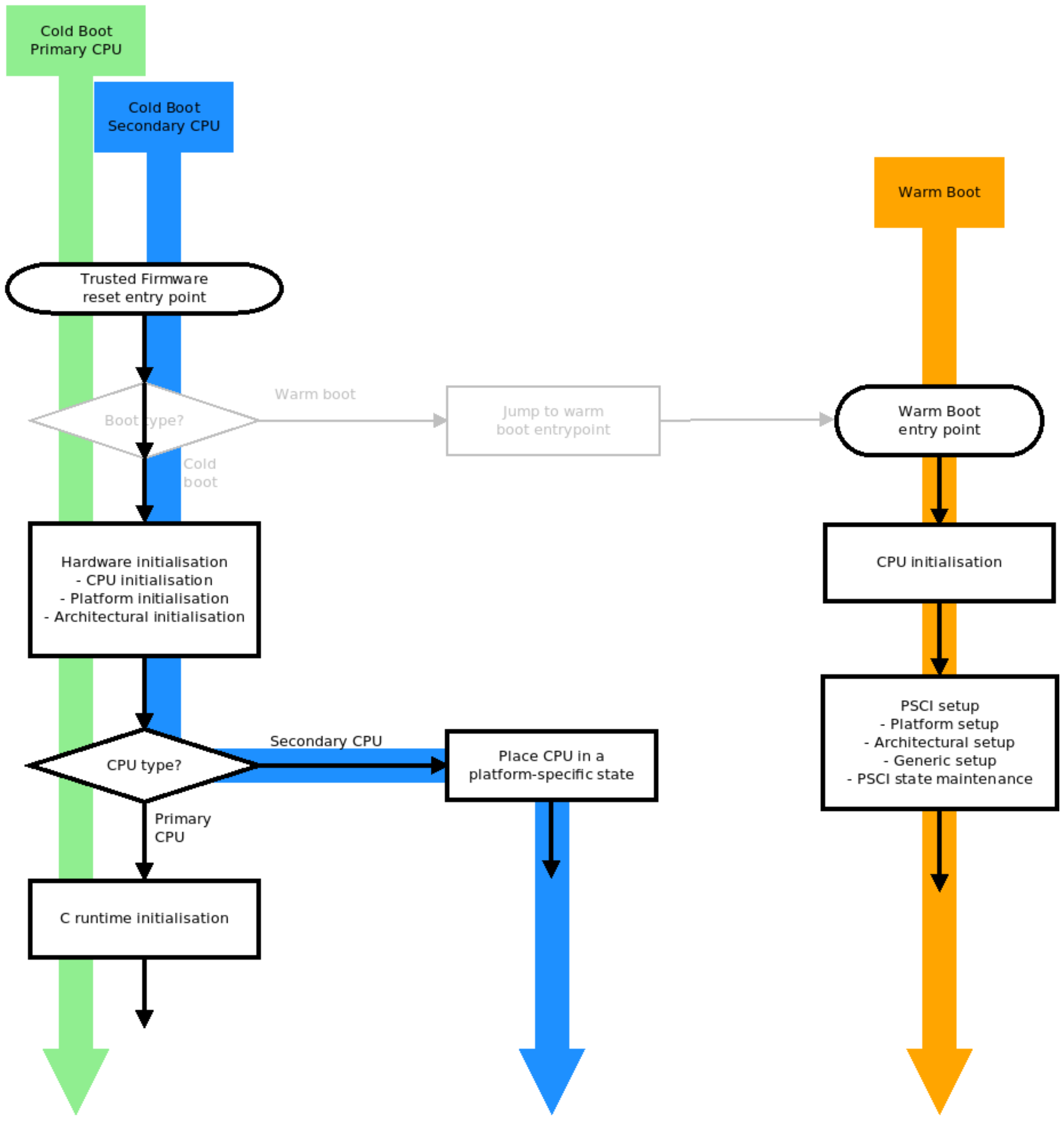
This diagram shows the default, unoptimised reset flow. Depending on the system configuration, some of these steps might be unnecessary. The following sections guide the platform integrator by indicating which build options exclude which steps, depending on the capability of the platform.

Note: If BL31 is used as the TF-A entry point instead of BL1, the diagram above is still relevant, as all these operations will occur in BL31 in this case. Please refer to section 6 “Using BL31 entrypoint as the reset address” for more information.

5.7.2 Programmable CPU reset address

By default, TF-A assumes that the CPU reset address is not programmable. Therefore, all CPUs start at the same address (typically address 0) whenever they reset. Further logic is then required to identify whether it is a cold or warm boot to direct CPUs to the right execution path.

If the reset vector address (reflected in the reset vector base address register `RVBAR_EL3`) is programmable then it is possible to make each CPU start directly at the right address, both on a cold and warm reset. Therefore, the boot type detection can be skipped, resulting in the following boot flow:



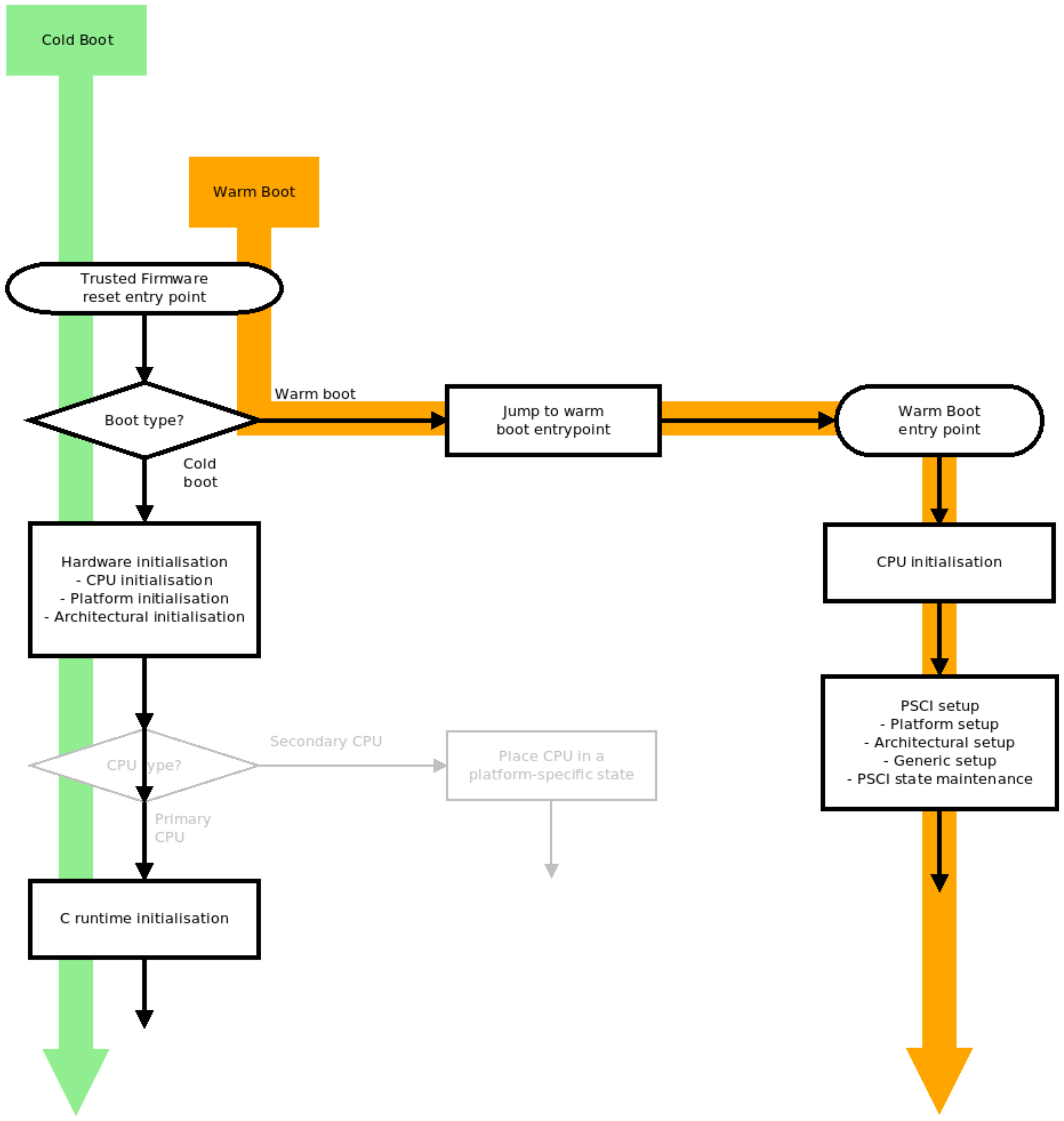
To enable this boot flow, compile TF-A with `PROGRAMMABLE_RESET_ADDRESS=1`. This option only affects the TF-A reset image, which is BL1 by default or BL31 if `RESET_TO_BL31=1`.

On both the FVP and Juno platforms, the reset vector address is not programmable so both ports use `PROGRAMMABLE_RESET_ADDRESS=0`.

5.7.3 Cold boot on a single CPU

By default, TF-A assumes that several CPUs may be released out of reset. Therefore, the cold boot code has to arbitrate access to hardware resources shared amongst CPUs. This is done by nominating one of the CPUs as the primary, which is responsible for initialising shared hardware and coordinating the boot flow with the other CPUs.

If the platform guarantees that only a single CPU will ever be brought up then no arbitration is required. The notion of primary/secondary CPU itself no longer applies. This results in the following boot flow:

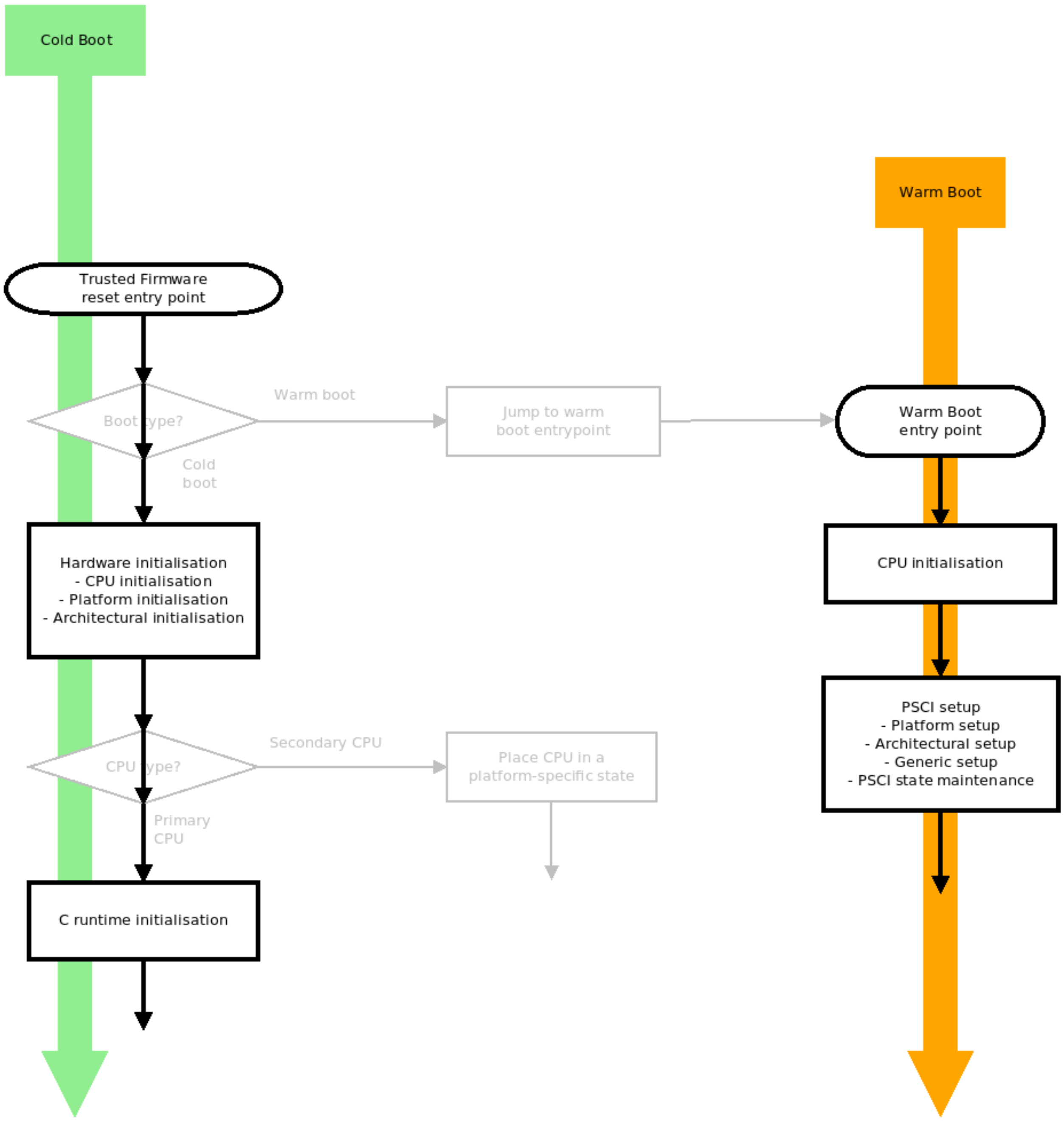


To enable this boot flow, compile TF-A with COLD_BOOT_SINGLE_CPU=1. This option only affects the TF-A reset image, which is BL1 by default or BL31 if RESET_TO_BL31=1.

On both the FVP and Juno platforms, although only one core is powered up by default, there are platform-specific ways to release any number of cores out of reset. Therefore, both platform ports use COLD_BOOT_SINGLE_CPU=0.

5.7.4 Programmable CPU reset address, Cold boot on a single CPU

It is obviously possible to combine both optimisations on platforms that have a programmable CPU reset address and which release a single CPU out of reset. This results in the following boot flow:



To enable this boot flow, compile TF-A with both COLD_BOOT_SINGLE_CPU=1 and

PROGRAMMABLE_RESET_ADDRESS=1. These options only affect the TF-A reset image, which is BL1 by default or BL31 if RESET_TO_BL31=1.

5.7.5 Using BL31 entrypoint as the reset address

On some platforms the runtime firmware (BL3x images) for the application processors are loaded by some firmware running on a secure system processor on the SoC, rather than by BL1 and BL2 running on the primary application processor. For this type of SoC it is desirable for the application processor to always reset to BL31 which eliminates the need for BL1 and BL2.

TF-A provides a build-time option RESET_TO_BL31 that includes some additional logic in the BL31 entry point to support this use case.

In this configuration, the platform's Trusted Boot Firmware must ensure that BL31 is loaded to its runtime address, which must match the CPU's RVBAR_EL3 reset vector base address, before the application processor is powered on. Additionally, platform software is responsible for loading the other BL3x images required and providing entry point information for them to BL31. Loading these images might be done by the Trusted Boot Firmware or by platform code in BL31.

Although the Arm FVP platform does not support programming the reset base address dynamically at run-time, it is possible to set the initial value of the RVBAR_EL3 register at start-up. This feature is provided on the Base FVP only.

It allows the Arm FVP port to support the RESET_TO_BL31 configuration, in which case the bl31.bin image must be loaded to its run address in Trusted SRAM and all CPU reset vectors be changed from the default 0x0 to this run address. See the *Arm Fixed Virtual Platforms (FVP)* for details of running the FVP models in this way.

Although technically it would be possible to program the reset base address with the right support in the SCP firmware, this is currently not implemented so the Juno port doesn't support the RESET_TO_BL31 configuration.

The RESET_TO_BL31 configuration requires some additions and changes in the BL31 functionality:

Determination of boot path

In this configuration, BL31 uses the same reset framework and code as the one described for BL1 above. Therefore, it is affected by the PROGRAMMABLE_RESET_ADDRESS and COLD_BOOT_SINGLE_CPU build options in the same way.

In the default, unoptimised BL31 reset flow, on a warm boot a CPU is directed to the PSCI implementation via a platform defined mechanism. On a cold boot, the platform must place any secondary CPUs into a safe state while the primary CPU executes a modified BL31 initialization, as described below.

Platform initialization

In this configuration, when the CPU resets to BL31 there are no parameters that can be passed in registers by previous boot stages. Instead, the platform code in BL31 needs to know, or be able to determine, the location of the BL32 (if required) and BL33 images and provide this information in response to the bl31_plat_get_next_image_ep_info() function.

Additionally, platform software is responsible for carrying out any security initialisation, for example programming a TrustZone address space controller. This might be done by the Trusted Boot Firmware or by platform code in BL31.

5.8 Trusted Board Boot

The Trusted Board Boot (TBB) feature prevents malicious firmware from running on the platform by authenticating all firmware images up to and including the normal world bootloader. It does this by establishing a Chain of Trust using Public-Key-Cryptography Standards (PKCS).

This document describes the design of Trusted Firmware-A (TF-A) TBB, which is an implementation of the Trusted Board Boot Requirements (TBBR) specification, Arm DEN0006D. It should be used in conjunction with the *Firmware Update (FWU)* design document, which implements a specific aspect of the TBBR.

5.8.1 Chain of Trust

A Chain of Trust (CoT) starts with a set of implicitly trusted components. On the Arm development platforms, these components are:

- A SHA-256 hash of the Root of Trust Public Key (ROTPK). It is stored in the trusted root-key storage registers. Alternatively, a development ROTPK might be used and its hash embedded into the BL1 and BL2 images (only for development purposes).
- The BL1 image, on the assumption that it resides in ROM so cannot be tampered with.

The remaining components in the CoT are either certificates or boot loader images. The certificates follow the X.509 v3 standard. This standard enables adding custom extensions to the certificates, which are used to store essential information to establish the CoT.

In the TBB CoT all certificates are self-signed. There is no need for a Certificate Authority (CA) because the CoT is not established by verifying the validity of a certificate's issuer but by the content of the certificate extensions. To sign the certificates, different signature schemes are available, please refer to the *Build Options* for more details.

The certificates are categorised as "Key" and "Content" certificates. Key certificates are used to verify public keys which have been used to sign content certificates. Content certificates are used to store the hash of a boot loader image. An image can be authenticated by calculating its hash and matching it with the hash extracted from the content certificate. Various hash algorithms are supported to calculate all hashes, please refer to the *Build Options* for more details.. The public keys and hashes are included as non-standard extension fields in the X.509 v3 certificates.

The keys used to establish the CoT are:

- **Root of trust key**

  The private part of this key is used to sign the BL2 content certificate and the trusted key certificate. The public part is the ROTPK.

- **Trusted world key**

  The private part is used to sign the key certificates corresponding to the secure world images (SCP_BL2, BL31 and BL32). The public part is stored in one of the extension fields in the trusted world certificate.

- **Non-trusted world key**

  The private part is used to sign the key certificate corresponding to the non secure world image (BL33). The public part is stored in one of the extension fields in the trusted world certificate.

- **BL3X keys**

  For each of SCP_BL2, BL31, BL32 and BL33, the private part is used to sign the content certificate for the BL3X image. The public part is stored in one of the extension fields in the corresponding key certificate.

The following images are included in the CoT:

- BL1

- BL2
- SCP_BL2 (optional)
- BL31
- BL33
- BL32 (optional)

The following certificates are used to authenticate the images.

- **BL2 content certificate**

  It is self-signed with the private part of the ROT key. It contains a hash of the BL2 image.

- **Trusted key certificate**

  It is self-signed with the private part of the ROT key. It contains the public part of the trusted world key and the public part of the non-trusted world key.

- **SCP_BL2 key certificate**

  It is self-signed with the trusted world key. It contains the public part of the SCP_BL2 key.

- **SCP_BL2 content certificate**

  It is self-signed with the SCP_BL2 key. It contains a hash of the SCP_BL2 image.

- **BL31 key certificate**

  It is self-signed with the trusted world key. It contains the public part of the BL31 key.

- **BL31 content certificate**

  It is self-signed with the BL31 key. It contains a hash of the BL31 image.

- **BL32 key certificate**

  It is self-signed with the trusted world key. It contains the public part of the BL32 key.

- **BL32 content certificate**

  It is self-signed with the BL32 key. It contains a hash of the BL32 image.

- **BL33 key certificate**

  It is self-signed with the non-trusted world key. It contains the public part of the BL33 key.

- **BL33 content certificate**

  It is self-signed with the BL33 key. It contains a hash of the BL33 image.

The SCP_BL2 and BL32 certificates are optional, but they must be present if the corresponding SCP_BL2 or BL32 images are present.

5.8.2 Trusted Board Boot Sequence

The CoT is verified through the following sequence of steps. The system panics if any of the steps fail.

- BL1 loads and verifies the BL2 content certificate. The issuer public key is read from the verified certificate. A hash of that key is calculated and compared with the hash of the ROTPK read from the trusted root-key storage registers. If they match, the BL2 hash is read from the certificate.

Note: The matching operation is platform specific and is currently unimplemented on the Arm development platforms.

- BL1 loads the BL2 image. Its hash is calculated and compared with the hash read from the certificate. Control is transferred to the BL2 image if all the comparisons succeed.
- BL2 loads and verifies the trusted key certificate. The issuer public key is read from the verified certificate. A hash of that key is calculated and compared with the hash of the ROTPK read from the trusted root-key storage registers. If the comparison succeeds, BL2 reads and saves the trusted and non-trusted world public keys from the verified certificate.

The next two steps are executed for each of the SCP_BL2, BL31 & BL32 images. The steps for the optional SCP_BL2 and BL32 images are skipped if these images are not present.

- BL2 loads and verifies the BL3x key certificate. The certificate signature is verified using the trusted world public key. If the signature verification succeeds, BL2 reads and saves the BL3x public key from the certificate.
- BL2 loads and verifies the BL3x content certificate. The signature is verified using the BL3x public key. If the signature verification succeeds, BL2 reads and saves the BL3x image hash from the certificate.

The next two steps are executed only for the BL33 image.

- BL2 loads and verifies the BL33 key certificate. If the signature verification succeeds, BL2 reads and saves the BL33 public key from the certificate.
- BL2 loads and verifies the BL33 content certificate. If the signature verification succeeds, BL2 reads and saves the BL33 image hash from the certificate.

The next step is executed for all the boot loader images.

- BL2 calculates the hash of each image. It compares it with the hash obtained from the corresponding content certificate. The image authentication succeeds if the hashes match.

The Trusted Board Boot implementation spans both generic and platform-specific BL1 and BL2 code, and in tool code on the host build machine. The feature is enabled through use of specific build flags as described in *Build Options*.

On the host machine, a tool generates the certificates, which are included in the FIP along with the boot loader images. These certificates are loaded in Trusted SRAM using the IO storage framework. They are then verified by an Authentication module included in TF-A.

The mechanism used for generating the FIP and the Authentication module are described in the following sections.

5.8.3 Authentication Framework

The authentication framework included in TF-A provides support to implement the desired trusted boot sequence. Arm platforms use this framework to implement the boot requirements specified in the Trusted Board Boot Requirements (TBBR) document.

More information about the authentication framework can be found in the *Authentication Framework & Chain of Trust* document.

5.8.4 Certificate Generation Tool

The `cert_create` tool is built and runs on the host machine as part of the TF-A build process when `GENERATE_COT=1`. It takes the boot loader images and keys as inputs (keys must be in PEM format) and generates the certificates (in DER format) required to establish the CoT. New keys can be generated by the tool in case they are not provided. The certificates are then passed as inputs to the `fiptool` utility for creating the FIP.

The certificates are also stored individually in the output build directory.

The tool resides in the `tools/cert_create` directory. It uses the OpenSSL SSL library version to generate the X.509 certificates. The specific version of the library that is required is given in the *Prerequisites* document.

Instructions for building and using the tool can be found at *Building the Certificate Generation Tool*.

5.8.5 Authenticated Encryption Framework

The authenticated encryption framework included in TF-A provides support to implement the optional firmware encryption feature. This feature can be optionally enabled on platforms to implement the optional requirement: R060_TBBR_FUNCTION as specified in the Trusted Board Boot Requirements (TBBR) document.

Note that due to security considerations and complexity of this feature, it is marked as experimental.

5.8.6 Firmware Encryption Tool

The `encrypt_fw` tool is built and runs on the host machine as part of the TF-A build process when `DECRYPTION_SUPPORT != none`. It takes the plain firmware image as input and generates the encrypted firmware image which can then be passed as input to the `fiptool` utility for creating the FIP.

The encrypted firmwares are also stored individually in the output build directory.

The tool resides in the `tools/encrypt_fw` directory. It uses OpenSSL SSL library version 1.0.1 or later to do authenticated encryption operation. Instructions for building and using the tool can be found in the *Building the Firmware Encryption Tool*.

5.9 Building FIP images with support for Trusted Board Boot

Trusted Board Boot primarily consists of the following two features:

- Image Authentication, described in *Trusted Board Boot*, and
- Firmware Update, described in *Firmware Update (FWU)*

The following steps should be followed to build FIP and (optionally) FWU_FIP images with support for these features:

1. Fulfill the dependencies of the `mbedtls` cryptographic and image parser modules by checking out a recent version of the mbed TLS Repository. It is important to use a version that is compatible with TF-A and fixes any known security vulnerabilities. See mbed TLS Security Center for more information. See the *Prerequisites* document for the appropriate version of mbed TLS to use.

   The `drivers/auth/mbedtls/mbedtls_*.mk` files contain the list of mbed TLS source files the modules depend upon. `include/drivers/auth/mbedtls/mbedtls_config.h` contains the configuration options required to build the mbed TLS sources.

Note that the mbed TLS library is licensed under the Apache version 2.0 license. Using mbed TLS source code will affect the licensing of TF-A binaries that are built using this library.

2. To build the FIP image, ensure the following command line variables are set while invoking `make` to build TF-A:

   - `MBEDTLS_DIR=<path of the directory containing mbed TLS sources>`
   - `TRUSTED_BOARD_BOOT=1`
   - `GENERATE_COT=1`

   By default, this will use the Chain of Trust described in the TBBR-client document. To select a different one, use the `COT` build option.

   In the case of Arm platforms, the location of the ROTPK hash must also be specified at build time. The following locations are currently supported (see `ARM_ROTPK_LOCATION` build option):

   - `ARM_ROTPK_LOCATION=regs`: the ROTPK hash is obtained from the Trusted root-key storage registers present in the platform. On Juno, these registers are read-only. On FVP Base and Cortex models, the registers are also read-only, but the value can be specified using the command line option `bp.trusted_key_storage.public_key` when launching the model. On Juno board, the default value corresponds to an ECDSA-SECP256R1 public key hash, whose private part is not currently available.
   - `ARM_ROTPK_LOCATION=devel_rsa`: use the default hash located in `plat/arm/board/common/rotpk/arm_rotpk_rsa_sha256.bin`. Enforce generation of the new hash if `ROT_KEY` is specified.
   - `ARM_ROTPK_LOCATION=devel_ecdsa`: use the default hash located in `plat/arm/board/common/rotpk/arm_rotpk_ecdsa_sha256.bin`. Enforce generation of the new hash if `ROT_KEY` is specified.

   Example of command line using RSA development keys:

   ```
   MBEDTLS_DIR=<path of the directory containing mbed TLS sources> \
   make PLAT=<platform> TRUSTED_BOARD_BOOT=1 GENERATE_COT=1        \
   ARM_ROTPK_LOCATION=devel_rsa                                    \
   ROT_KEY=plat/arm/board/common/rotpk/arm_rotprivk_rsa.pem        \
   BL33=<path-to>/<bl33_image>                                     \
   all fip
   ```

   The result of this build will be the bl1.bin and the fip.bin binaries. This FIP will include the certificates corresponding to the selected Chain of Trust. These certificates can also be found in the output build directory.

3. The optional FWU_FIP contains any additional images to be loaded from Non-Volatile storage during the *Firmware Update (FWU)* process. To build the FWU_FIP, any FWU images required by the platform must be specified on the command line. On Arm development platforms like Juno, these are:

   - NS_BL2U. The AP non-secure Firmware Updater image.
   - SCP_BL2U. The SCP Firmware Update Configuration image.

   Example of Juno command line for generating both `fwu` and `fwu_fip` targets using RSA development:

   ```
   MBEDTLS_DIR=<path of the directory containing mbed TLS sources> \
   make PLAT=juno TRUSTED_BOARD_BOOT=1 GENERATE_COT=1              \
   ARM_ROTPK_LOCATION=devel_rsa                                    \
   ROT_KEY=plat/arm/board/common/rotpk/arm_rotprivk_rsa.pem        \
   BL33=<path-to>/<bl33_image>                                     \
   SCP_BL2=<path-to>/<scp_bl2_image>                               \
   SCP_BL2U=<path-to>/<scp_bl2u_image>                             \
   ```

(continues on next page)

(continued from previous page)

```
NS_BL2U=<path-to>/<ns_bl2u_image>                              \
all fip fwu_fip
```

Note: The BL2U image will be built by default and added to the FWU_FIP. The user may override this by adding `BL2U=<path-to>/<bl2u_image>` to the command line above.

Note: Building and installing the non-secure and SCP FWU images (NS_BL1U, NS_BL2U and SCP_BL2U) is outside the scope of this document.

The result of this build will be bl1.bin, fip.bin and fwu_fip.bin binaries. Both the FIP and FWU_FIP will include the certificates corresponding to the selected Chain of Trust. These certificates can also be found in the output build directory.

CHAPTER

SIX

PLATFORM PORTS

6.1 Allwinner ARMv8 SoCs

Trusted Firmware-A (TF-A) implements the EL3 firmware layer for Allwinner SoCs with ARMv8 cores. Only BL31 is used to provide proper EL3 setup and PSCI runtime services.

U-Boot's SPL acts as a loader, loading both BL31 and BL33 (typically U-Boot). Loading is done from SD card, eMMC or SPI flash, also via an USB debug interface (FEL).

BL31 lives in SRAM A2, which is documented to be accessible from secure world only.

Current limitations:

- Missing PMIC support

After building bl31.bin, the binary must be fed to the U-Boot build system to include it in the FIT image that the SPL loader will process. bl31.bin can be either copied (or sym-linked) into U-Boot's root directory, or the environment variable BL31 must contain the binary's path. See the respective U-Boot documentation for more details.

To build for machines with an A64 or H5 SoC:

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=sun50i_a64 DEBUG=1 bl31
```

To build for machines with an H6 SoC:

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=sun50i_h6 DEBUG=1 bl31
```

6.1.1 Trusted OS dispatcher

One can boot Trusted OS(OP-TEE OS, bl32 image) along side bl31 image on Allwinner A64.

In order to include the 'opteed' dispatcher in the image, pass 'SPD=opteed' on the command line while compiling the bl31 image and make sure the loader (SPL) loads the Trusted OS binary to the beginning of DRAM (0x40000000).

6.2 Arm Development Platforms

6.2.1 Arm Juno Development Platform

Platform-specific build options

- `JUNO_TZMP1` : Boolean option to configure Juno to be used for TrustZone Media Protection (TZ-MP1). Default value of this flag is 0.

Running software on Juno

This version of TF-A has been tested on variants r0, r1 and r2 of Juno.

To execute the software stack on Juno, the version of the Juno board recovery image indicated in the Linaro Release Notes must be installed. If you have an earlier version installed or are unsure which version is installed, please re-install the recovery image by following the Instructions for using Linaro's deliverables on Juno.

Preparing TF-A images

After building TF-A, the files `bl1.bin` and `fip.bin` need copying to the `SOFTWARE/` directory of the Juno SD card.

Creating a Firmware Image Package (FIP)

This section provides Juno and FVP specific instructions to build Trusted Firmware, obtain the additional required firmware, and pack it all together in a single FIP binary. It assumes that a Linaro release has been installed.

Note: Pre-built binaries for AArch32 are available from Linaro Release 16.12 onwards. Before that release, pre-built binaries are only available for AArch64.

Warning: Follow the full instructions for one platform before switching to a different one. Mixing instructions for different platforms may result in corrupted binaries.

Warning: The uboot image downloaded by the Linaro workspace script does not always match the uboot image packaged as BL33 in the corresponding fip file. It is recommended to use the version that is packaged in the fip file using the instructions below.

Note: For the FVP, the kernel FDT is packaged in FIP during build and loaded by the firmware at runtime.

1. Clean the working directory

   ```
   make realclean
   ```

2. Obtain SCP_BL2 (Juno) and BL33 (all platforms)

   Use the fiptool to extract the SCP_BL2 and BL33 images from the FIP package included in the Linaro release:

```
# Build the fiptool
make [DEBUG=1] [V=1] fiptool

# Unpack firmware images from Linaro FIP
./tools/fiptool/fiptool unpack <path-to-linaro-release>/[SOFTWARE]/fip.bin
```

The unpack operation will result in a set of binary images extracted to the current working directory. The SCP_BL2 image corresponds to `scp-fw.bin` and BL33 corresponds to `nt-fw.bin`.

Note: The fiptool will complain if the images to be unpacked already exist in the current directory. If that is the case, either delete those files or use the `--force` option to overwrite.

Note: For AArch32, the instructions below assume that nt-fw.bin is a normal world boot loader that supports AArch32.

---

3. Build TF-A images and create a new FIP for FVP

```
# AArch64
make PLAT=fvp BL33=nt-fw.bin all fip

# AArch32
make PLAT=fvp ARCH=aarch32 AARCH32_SP=sp_min BL33=nt-fw.bin all fip
```

4. Build TF-A images and create a new FIP for Juno

For AArch64:

Building for AArch64 on Juno simply requires the addition of `SCP_BL2` as a build parameter.

```
make PLAT=juno BL33=nt-fw.bin SCP_BL2=scp-fw.bin all fip
```

For AArch32:

Hardware restrictions on Juno prevent cold reset into AArch32 execution mode, therefore BL1 and BL2 must be compiled for AArch64, and BL32 is compiled separately for AArch32.

- Before building BL32, the environment variable `CROSS_COMPILE` must point to the AArch32 Linaro cross compiler.

```
export CROSS_COMPILE=<path-to-aarch32-gcc>/bin/arm-linux-gnueabihf-
```

- Build BL32 in AArch32.

```
make ARCH=aarch32 PLAT=juno AARCH32_SP=sp_min \
RESET_TO_SP_MIN=1 JUNO_AARCH32_EL3_RUNTIME=1 bl32
```

- Save `bl32.bin` to a temporary location and clean the build products.

```
cp <path-to-build>/bl32.bin <path-to-temporary>
make realclean
```

- Before building BL1 and BL2, the environment variable `CROSS_COMPILE` must point to the AArch64 Linaro cross compiler.

```
export CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-none-elf-
```

- The following parameters should be used to build BL1 and BL2 in AArch64 and point to the BL32 file.

```
make ARCH=aarch64 PLAT=juno JUNO_AARCH32_EL3_RUNTIME=1 \
BL33=nt-fw.bin SCP_BL2=scp-fw.bin \
BL32=<path-to-temporary>/bl32.bin all fip
```

The resulting BL1 and FIP images may be found in:

```
# Juno
./build/juno/release/bl1.bin
./build/juno/release/fip.bin

# FVP
./build/fvp/release/bl1.bin
./build/fvp/release/fip.bin
```

Booting Firmware Update images

The new images must be programmed in flash memory by adding an entry in the `SITE1/HBI0262x/images.txt` configuration file on the Juno SD card (where `x` depends on the revision of the Juno board). Refer to the Juno Getting Started Guide, section 2.3 “Flash memory programming” for more information. User should ensure these do not overlap with any other entries in the file.

```
NOR10UPDATE: AUTO                       ;Image Update:NONE/AUTO/FORCE
NOR10ADDRESS: 0x00400000                ;Image Flash Address [ns_bl2u_base_address]
NOR10FILE: \SOFTWARE\fwu_fip.bin        ;Image File Name
NOR10LOAD: 00000000                     ;Image Load Address
NOR10ENTRY: 00000000                    ;Image Entry Point

NOR11UPDATE: AUTO                       ;Image Update:NONE/AUTO/FORCE
NOR11ADDRESS: 0x03EB8000                ;Image Flash Address [ns_bl1u_base_address]
NOR11FILE: \SOFTWARE\ns_bl1u.bin        ;Image File Name
NOR11LOAD: 00000000                     ;Image Load Address
```

The address ns_bl1u_base_address is the value of NS_BL1U_BASE - 0x8000000. In the same way, the address ns_bl2u_base_address is the value of NS_BL2U_BASE - 0x8000000.

Booting an EL3 payload

If the EL3 payload is able to execute in place, it may be programmed in flash memory by adding an entry in the `SITE1/HBI0262x/images.txt` configuration file on the Juno SD card (where `x` depends on the revision of the Juno board). Refer to the Juno Getting Started Guide, section 2.3 “Flash memory programming” for more information.

Alternatively, the same DS-5 command mentioned in the FVP section above can be used to load the EL3 payload’s ELF file over JTAG on Juno.

For more information on EL3 payloads in general, see *Booting an EL3 payload*.

Booting a preloaded kernel image

The Trusted Firmware must be compiled in a similar way as for FVP explained above. The process to load binaries to memory is the one explained in *plat_juno_booting_el3_payload*.

Testing System Suspend

The SYSTEM SUSPEND is a PSCI API which can be used to implement system suspend to RAM. For more details refer to section 5.16 of PSCI. To test system suspend on Juno, at the linux shell prompt, issue the following command:

```
echo +10 > /sys/class/rtc/rtc0/wakealarm
echo -n mem > /sys/power/state
```

The Juno board should suspend to RAM and then wakeup after 10 seconds due to wakeup interrupt from RTC.

Additional Resources

Please visit the Arm Platforms Portal to get support and obtain any other Juno software information. Please also refer to the Juno Getting Started Guide to get more detailed information about the Juno Arm development platform and how to configure it.

6.2.2 Arm Fixed Virtual Platforms (FVP)

Fixed Virtual Platform (FVP) Support

This section lists the supported Arm *FVP* platforms. Please refer to the FVP documentation for a detailed description of the model parameter options.

The latest version of the AArch64 build of TF-A has been tested on the following Arm FVPs without shifted affinities, and that do not support threaded CPU cores (64-bit host machine only).

Note: The FVP models used are Version 11.12 Build 38, unless otherwise stated.

- `FVP_Base_AEMvA`
- `FVP_Base_AEMv8A-AEMv8A`
- `FVP_Base_AEMv8A-AEMv8A-AEMv8A-AEMv8A-CCN502`
- `FVP_Base_RevC-2xAEMv8A`
- `FVP_Base_Cortex-A32x4`
- `FVP_Base_Cortex-A35x4`
- `FVP_Base_Cortex-A53x4`
- `FVP_Base_Cortex-A55x4+Cortex-A75x4`
- `FVP_Base_Cortex-A55x4`
- `FVP_Base_Cortex-A57x1-A53x1`

- `FVP_Base_Cortex-A57x2-A53x4`
- `FVP_Base_Cortex-A57x4-A53x4`
- `FVP_Base_Cortex-A57x4`
- `FVP_Base_Cortex-A65x4`
- `FVP_Base_Cortex-A65AEx8`
- `FVP_Base_Cortex-A72x4-A53x4`
- `FVP_Base_Cortex-A72x4`
- `FVP_Base_Cortex-A73x4-A53x4`
- `FVP_Base_Cortex-A73x4`
- `FVP_Base_Cortex-A75x4`
- `FVP_Base_Cortex-A76x4`
- `FVP_Base_Cortex-A76AEx4`
- `FVP_Base_Cortex-A76AEx8`
- `FVP_Base_Cortex-A77x4`
- `FVP_Base_Cortex-A78x4`
- `FVP_Base_Neoverse-E1x1`
- `FVP_Base_Neoverse-E1x2`
- `FVP_Base_Neoverse-E1x4`
- `FVP_Base_Neoverse-N1x4`
- `FVP_Base_Neoverse-V1x4`
- `FVP_CSS_SGI-575` (Version 11.10 build 36)
- `FVP_CSS_SGM-775`
- `FVP_RD_E1_edge` (Version 11.9 build 41)
- `FVP_RD_N1_edge` (Version 11.10 build 36)
- `FVP_RD_N1_edge_dual` (Version 11.10 build 36)
- `FVP_RD_Daniel` (Version 11.10 build 36)
- `FVP_TC0` (Version 0.0 build 6114)
- `Foundation_Platform`

The latest version of the AArch32 build of TF-A has been tested on the following Arm FVPs without shifted affinities, and that do not support threaded CPU cores (64-bit host machine only).

- `FVP_Base_AEMvA`
- `FVP_Base_AEMv8A-AEMv8A`
- `FVP_Base_Cortex-A32x4`

Note: The `FVP_Base_RevC-2xAEMv8A` FVP only supports shifted affinities, which is not compatible with legacy GIC configurations. Therefore this FVP does not support these legacy GIC configurations.

The *Foundation* and *Base* FVPs can be downloaded free of charge. See the Arm FVP website. The Cortex-A models listed above are also available to download from Arm's website.

Note: The build numbers quoted above are those reported by launching the FVP with the `--version` parameter.

Note: Linaro provides a ramdisk image in prebuilt FVP configurations and full file systems that can be downloaded separately. To run an FVP with a virtio file system image an additional FVP configuration option `-C bp.virtioblockdevice.image_path="<path-to>/<file-system-image>` can be used.

Note: The software will not work on Version 1.0 of the Foundation FVP. The commands below would report an `unhandled argument` error in this case.

Note: FVPs can be launched with `--cadi-server` option such that a CADI-compliant debugger (for example, Arm DS-5) can connect to and control its execution.

Warning: Since FVP model Version 11.0 Build 11.0.34 and Version 8.5 Build 0.8.5202 the internal synchronisation timings changed compared to older versions of the models. The models can be launched with `-Q 100` option if they are required to match the run time characteristics of the older versions.

All the above platforms have been tested with Linaro Release 19.06.

Arm FVP Platform Specific Build Options

- `FVP_CLUSTER_COUNT` : Configures the cluster count to be used to build the topology tree within TF-A. By default TF-A is configured for dual cluster topology and this option can be used to override the default value.
- `FVP_INTERCONNECT_DRIVER`: Selects the interconnect driver to be built. The default interconnect driver depends on the value of `FVP_CLUSTER_COUNT` as explained in the options below:
  - `FVP_CCI` : The CCI driver is selected. This is the default if 0 < `FVP_CLUSTER_COUNT` <= 2.
  - `FVP_CCN` : The CCN driver is selected. This is the default if `FVP_CLUSTER_COUNT` > 2.
- `FVP_MAX_CPUS_PER_CLUSTER`: Sets the maximum number of CPUs implemented in a single cluster. This option defaults to 4.
- `FVP_MAX_PE_PER_CPU`: Sets the maximum number of PEs implemented on any CPU in the system. This option defaults to 1. Note that the build option `ARM_PLAT_MT` doesn't have any effect on FVP platforms.
- `FVP_USE_GIC_DRIVER` : Selects the GIC driver to be built. Options:
  - `FVP_GICV2` : The GICv2 only driver is selected
  - `FVP_GICV3` : The GICv3 only driver is selected (default option)
- `FVP_HW_CONFIG_DTS` : Specify the path to the DTS file to be compiled to DTB and packaged in FIP as the HW_CONFIG. See *Firmware Design* for details on HW_CONFIG. By default, this is initialized to a sensible DTS file in `fdts/` folder depending on other build options. But some cases, like shifted affinity format for MPIDR, cannot be detected at build time and this option is needed to specify the appropriate DTS file.

- FVP_HW_CONFIG : Specify the path to the HW_CONFIG blob to be packaged in FIP. See *Firmware Design* for details on HW_CONFIG. This option is similar to the FVP_HW_CONFIG_DTS option, but it directly specifies the HW_CONFIG blob instead of the DTS file. This option is useful to override the default HW_CONFIG selected by the build system.

Booting Firmware Update images

When Firmware Update (FWU) is enabled there are at least 2 new images that have to be loaded, the Non-Secure FWU ROM (NS-BL1U), and the FWU FIP.

The additional fip images must be loaded with:

```
--data cluster0.cpu0="<path_to>/ns_bl1u.bin"@0x0beb8000     [ns_bl1u_base_address]
--data cluster0.cpu0="<path_to>/fwu_fip.bin"@0x08400000     [ns_bl2u_base_address]
```

The address ns_bl1u_base_address is the value of NS_BL1U_BASE. In the same way, the address ns_bl2u_base_address is the value of NS_BL2U_BASE.

Booting an EL3 payload

The EL3 payloads boot flow requires the CPU's mailbox to be cleared at reset for the secondary CPUs holding pen to work properly. Unfortunately, its reset value is undefined on the FVP platform and the FVP platform code doesn't clear it. Therefore, one must modify the way the model is normally invoked in order to clear the mailbox at start-up.

One way to do that is to create an 8-byte file containing all zero bytes using the following command:

```
dd if=/dev/zero of=mailbox.dat bs=1 count=8
```

and pre-load it into the FVP memory at the mailbox address (i.e. 0x04000000) using the following model parameters:

```
--data cluster0.cpu0=mailbox.dat@0x04000000   [Base FVPs]
--data=mailbox.dat@0x04000000                 [Foundation FVP]
```

To provide the model with the EL3 payload image, the following methods may be used:

1. If the EL3 payload is able to execute in place, it may be programmed into flash memory. On Base Cortex and AEM FVPs, the following model parameter loads it at the base address of the NOR FLASH1 (the NOR FLASH0 is already used for the FIP):

   ```
   -C bp.flashloader1.fname="<path-to>/<el3-payload>"
   ```

   On Foundation FVP, there is no flash loader component and the EL3 payload may be programmed anywhere in flash using method 3 below.

2. When using the SPIN_ON_BL1_EXIT=1 loading method, the following DS-5 command may be used to load the EL3 payload ELF image over JTAG:

   ```
   load <path-to>/el3-payload.elf
   ```

3. The EL3 payload may be pre-loaded in volatile memory using the following model parameters:

   ```
   --data cluster0.cpu0="<path-to>/el3-payload>"@address   [Base FVPs]
   --data="<path-to>/<el3-payload>"@address                [Foundation FVP]
   ```

   The address provided to the FVP must match the EL3_PAYLOAD_BASE address used when building TF-A.

Booting a preloaded kernel image (Base FVP)

The following example uses a simplified boot flow by directly jumping from the TF-A to the Linux kernel, which will use a ramdisk as filesystem. This can be useful if both the kernel and the device tree blob (DTB) are already present in memory (like in FVP).

For example, if the kernel is loaded at `0x80080000` and the DTB is loaded at address `0x82000000`, the firmware can be built like this:

```
CROSS_COMPILE=aarch64-none-elf-  \
make PLAT=fvp DEBUG=1             \
RESET_TO_BL31=1                   \
ARM_LINUX_KERNEL_AS_BL33=1        \
PRELOADED_BL33_BASE=0x80080000    \
ARM_PRELOADED_DTB_BASE=0x82000000 \
all fip
```

Now, it is needed to modify the DTB so that the kernel knows the address of the ramdisk. The following script generates a patched DTB from the provided one, assuming that the ramdisk is loaded at address `0x84000000`. Note that this script assumes that the user is using a ramdisk image prepared for U-Boot, like the ones provided by Linaro. If using a ramdisk without this header,the `0x40` offset in `INITRD_START` has to be removed.

```
#!/bin/bash

# Path to the input DTB
KERNEL_DTB=<path-to>/<fdt>
# Path to the output DTB
PATCHED_KERNEL_DTB=<path-to>/<patched-fdt>
# Base address of the ramdisk
INITRD_BASE=0x84000000
# Path to the ramdisk
INITRD=<path-to>/<ramdisk.img>

# Skip uboot header (64 bytes)
INITRD_START=$(printf "0x%x" $((${INITRD_BASE} + 0x40)) )
INITRD_SIZE=$(stat -Lc %s ${INITRD})
INITRD_END=$(printf "0x%x" $((${INITRD_BASE} + ${INITRD_SIZE})) )

CHOSEN_NODE=$(echo                                      \
"/ {                                                    \
        chosen {                                        \
                linux,initrd-start = <${INITRD_START}>; \
                linux,initrd-end = <${INITRD_END}>;     \
        };                                              \
};")

echo $(dtc -O dts -I dtb ${KERNEL_DTB}) ${CHOSEN_NODE} |  \
        dtc -O dtb -o ${PATCHED_KERNEL_DTB} -
```

And the FVP binary can be run with the following command:

```
<path-to>/FVP_Base_AEMv8A-AEMv8A                          \
-C pctl.startup=0.0.0.0                                   \
-C bp.secure_memory=1                                     \
-C cluster0.NUM_CORES=4                                   \
-C cluster1.NUM_CORES=4                                   \
-C cache_state_modelled=1                                 \
-C cluster0.cpu0.RVBAR=0x04001000                         \
```

(continues on next page)

(continued from previous page)

```
-C cluster0.cpu1.RVBAR=0x04001000                           \
-C cluster0.cpu2.RVBAR=0x04001000                           \
-C cluster0.cpu3.RVBAR=0x04001000                           \
-C cluster1.cpu0.RVBAR=0x04001000                           \
-C cluster1.cpu1.RVBAR=0x04001000                           \
-C cluster1.cpu2.RVBAR=0x04001000                           \
-C cluster1.cpu3.RVBAR=0x04001000                           \
--data cluster0.cpu0="<path-to>/bl31.bin"@0x04001000        \
--data cluster0.cpu0="<path-to>/<patched-fdt>"@0x82000000   \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000 \
--data cluster0.cpu0="<path-to>/<ramdisk.img>"@0x84000000
```

Obtaining the Flattened Device Trees

Depending on the FVP configuration and Linux configuration used, different FDT files are required. FDT source files for the Foundation and Base FVPs can be found in the TF-A source directory under `fdts/`. The Foundation FVP has a subset of the Base FVP components. For example, the Foundation FVP lacks CLCD and MMC support, and has only one CPU cluster.

Note: It is not recommended to use the FDTs built along the kernel because not all FDTs are available from there.

The dynamic configuration capability is enabled in the firmware for FVPs. This means that the firmware can authenticate and load the FDT if present in FIP. A default FDT is packaged into FIP during the build based on the build configuration. This can be overridden by using the `FVP_HW_CONFIG` or `FVP_HW_CONFIG_DTS` build options (refer to *Arm FVP Platform Specific Build Options* for details on the options).

- `fvp-base-gicv2-psci.dts`

  For use with models such as the Cortex-A57-A53 Base FVPs without shifted affinities and with Base memory map configuration.

- `fvp-base-gicv2-psci-aarch32.dts`

  For use with models such as the Cortex-A32 Base FVPs without shifted affinities and running Linux in AArch32 state with Base memory map configuration.

- `fvp-base-gicv3-psci.dts`

  For use with models such as the Cortex-A57-A53 Base FVPs without shifted affinities and with Base memory map configuration and Linux GICv3 support.

- `fvp-base-gicv3-psci-1t.dts`

  For use with models such as the AEMv8-RevC Base FVP with shifted affinities, single threaded CPUs, Base memory map configuration and Linux GICv3 support.

- `fvp-base-gicv3-psci-dynamiq.dts`

  For use with models as the Cortex-A55-A75 Base FVPs with shifted affinities, single cluster, single threaded CPUs, Base memory map configuration and Linux GICv3 support.

- `fvp-base-gicv3-psci-aarch32.dts`

  For use with models such as the Cortex-A32 Base FVPs without shifted affinities and running Linux in AArch32 state with Base memory map configuration and Linux GICv3 support.

- `fvp-foundation-gicv2-psci.dts`

For use with Foundation FVP with Base memory map configuration.

- `fvp-foundation-gicv3-psci.dts`

  (Default) For use with Foundation FVP with Base memory map configuration and Linux GICv3 support.

Running on the Foundation FVP with reset to BL1 entrypoint

The following `Foundation_Platform` parameters should be used to boot Linux with 4 CPUs using the AArch64 build of TF-A.

```
<path-to>/Foundation_Platform                   \
--cores=4                                       \
--arm-v8.0                                      \
--secure-memory                                 \
--visualization                                 \
--gicv3                                         \
--data="<path-to>/<bl1-binary>"@0x0             \
--data="<path-to>/<FIP-binary>"@0x08000000      \
--data="<path-to>/<kernel-binary>"@0x80080000   \
--data="<path-to>/<ramdisk-binary>"@0x84000000
```

Notes:

- BL1 is loaded at the start of the Trusted ROM.
- The Firmware Image Package is loaded at the start of NOR FLASH0.
- The firmware loads the FDT packaged in FIP to the DRAM. The FDT load address is specified via the `hw_config_addr` property in TB_FW_CONFIG for FVP.
- The default use-case for the Foundation FVP is to use the `--gicv3` option and enable the GICv3 device in the model. Note that without this option, the Foundation FVP defaults to legacy (Versatile Express) memory map which is not supported by TF-A.
- In order for TF-A to run correctly on the Foundation FVP, the architecture versions must match. The Foundation FVP defaults to the highest v8.x version it supports but the default build for TF-A is for v8.0. To avoid issues either start the Foundation FVP to use v8.0 architecture using the `--arm-v8.0` option, or build TF-A with an appropriate value for `ARM_ARCH_MINOR`.

Running on the AEMv8 Base FVP with reset to BL1 entrypoint

The following `FVP_Base_RevC-2xAEMv8A` parameters should be used to boot Linux with 8 CPUs using the AArch64 build of TF-A.

```
<path-to>/FVP_Base_RevC-2xAEMv8A                            \
-C pctl.startup=0.0.0.0                                     \
-C bp.secure_memory=1                                       \
-C bp.tzc_400.diagnostics=1                                 \
-C cluster0.NUM_CORES=4                                     \
-C cluster1.NUM_CORES=4                                     \
-C cache_state_modelled=1                                   \
-C bp.secureflashloader.fname="<path-to>/<bl1-binary>"      \
-C bp.flashloader0.fname="<path-to>/<FIP-binary>"           \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000 \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Note: The `FVP_Base_RevC-2xAEMv8A` has shifted affinities and requires a specific DTS for all the CPUs to be loaded.

Running on the AEMv8 Base FVP (AArch32) with reset to BL1 entrypoint

The following `FVP_Base_AEMv8A-AEMv8A` parameters should be used to boot Linux with 8 CPUs using the AArch32 build of TF-A.

```
<path-to>/FVP_Base_AEMv8A-AEMv8A                              \
-C pctl.startup=0.0.0.0                                       \
-C bp.secure_memory=1                                         \
-C bp.tzc_400.diagnostics=1                                   \
-C cluster0.NUM_CORES=4                                       \
-C cluster1.NUM_CORES=4                                       \
-C cache_state_modelled=1                                     \
-C cluster0.cpu0.CONFIG64=0                                   \
-C cluster0.cpu1.CONFIG64=0                                   \
-C cluster0.cpu2.CONFIG64=0                                   \
-C cluster0.cpu3.CONFIG64=0                                   \
-C cluster1.cpu0.CONFIG64=0                                   \
-C cluster1.cpu1.CONFIG64=0                                   \
-C cluster1.cpu2.CONFIG64=0                                   \
-C cluster1.cpu3.CONFIG64=0                                   \
-C bp.secureflashloader.fname="<path-to>/<bl1-binary>"        \
-C bp.flashloader0.fname="<path-to>/<FIP-binary>"             \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000 \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Running on the Cortex-A57-A53 Base FVP with reset to BL1 entrypoint

The following `FVP_Base_Cortex-A57x4-A53x4` model parameters should be used to boot Linux with 8 CPUs using the AArch64 build of TF-A.

```
<path-to>/FVP_Base_Cortex-A57x4-A53x4                         \
-C pctl.startup=0.0.0.0                                       \
-C bp.secure_memory=1                                         \
-C bp.tzc_400.diagnostics=1                                   \
-C cache_state_modelled=1                                     \
-C bp.secureflashloader.fname="<path-to>/<bl1-binary>"        \
-C bp.flashloader0.fname="<path-to>/<FIP-binary>"             \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000 \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Running on the Cortex-A32 Base FVP (AArch32) with reset to BL1 entrypoint

The following `FVP_Base_Cortex-A32x4` model parameters should be used to boot Linux with 4 CPUs using the AArch32 build of TF-A.

```
<path-to>/FVP_Base_Cortex-A32x4                                 \
-C pctl.startup=0.0.0.0                                         \
-C bp.secure_memory=1                                           \
-C bp.tzc_400.diagnostics=1                                     \
-C cache_state_modelled=1                                       \
-C bp.secureflashloader.fname="<path-to>/<bl1-binary>"          \
-C bp.flashloader0.fname="<path-to>/<FIP-binary>"               \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000 \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Running on the AEMv8 Base FVP with reset to BL31 entrypoint

The following `FVP_Base_RevC-2xAEMv8A` parameters should be used to boot Linux with 8 CPUs using the AArch64 build of TF-A.

```
<path-to>/FVP_Base_RevC-2xAEMv8A                                 \
-C pctl.startup=0.0.0.0                                          \
-C bp.secure_memory=1                                            \
-C bp.tzc_400.diagnostics=1                                      \
-C cluster0.NUM_CORES=4                                          \
-C cluster1.NUM_CORES=4                                          \
-C cache_state_modelled=1                                        \
-C cluster0.cpu0.RVBAR=0x04010000                                \
-C cluster0.cpu1.RVBAR=0x04010000                                \
-C cluster0.cpu2.RVBAR=0x04010000                                \
-C cluster0.cpu3.RVBAR=0x04010000                                \
-C cluster1.cpu0.RVBAR=0x04010000                                \
-C cluster1.cpu1.RVBAR=0x04010000                                \
-C cluster1.cpu2.RVBAR=0x04010000                                \
-C cluster1.cpu3.RVBAR=0x04010000                                \
--data cluster0.cpu0="<path-to>/<bl31-binary>"@0x04010000    \
--data cluster0.cpu0="<path-to>/<bl32-binary>"@0xff000000    \
--data cluster0.cpu0="<path-to>/<bl33-binary>"@0x88000000    \
--data cluster0.cpu0="<path-to>/<fdt>"@0x82000000            \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000  \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Notes:

- If Position Independent Executable (PIE) support is enabled for BL31 in this config, it can be loaded at any valid address for execution.
- Since a FIP is not loaded when using BL31 as reset entrypoint, the `--data="<path-to><bl31|bl32|bl33-binary>"@<base-address-of-binary>` parameter is needed to load the individual bootloader images in memory. BL32 image is only needed if BL31 has been built to expect a Secure-EL1 Payload. For the same reason, the FDT needs to be compiled from the DT source and loaded via the `--data cluster0.cpu0="<path-to>/<fdt>"@0x82000000` parameter.
- The `FVP_Base_RevC-2xAEMv8A` has shifted affinities and requires a specific DTS for all the CPUs to be loaded.

- The `-C cluster<X>.cpu<Y>.RVBAR=@<base-address-of-bl31>` parameter, where X and Y are the cluster and CPU numbers respectively, is used to set the reset vector for each core.
- Changing the default value of `ARM_TSP_RAM_LOCATION` will also require changing the value of `--data="<path-to><bl32-binary>"@<base-address-of-bl32>` to the new value of `BL32_BASE`.

Running on the AEMv8 Base FVP (AArch32) with reset to SP_MIN entrypoint

The following `FVP_Base_AEMv8A-AEMv8A` parameters should be used to boot Linux with 8 CPUs using the AArch32 build of TF-A.

```
<path-to>/FVP_Base_AEMv8A-AEMv8A                          \
-C pctl.startup=0.0.0.0                                   \
-C bp.secure_memory=1                                     \
-C bp.tzc_400.diagnostics=1                               \
-C cluster0.NUM_CORES=4                                   \
-C cluster1.NUM_CORES=4                                   \
-C cache_state_modelled=1                                 \
-C cluster0.cpu0.CONFIG64=0                               \
-C cluster0.cpu1.CONFIG64=0                               \
-C cluster0.cpu2.CONFIG64=0                               \
-C cluster0.cpu3.CONFIG64=0                               \
-C cluster1.cpu0.CONFIG64=0                               \
-C cluster1.cpu1.CONFIG64=0                               \
-C cluster1.cpu2.CONFIG64=0                               \
-C cluster1.cpu3.CONFIG64=0                               \
-C cluster0.cpu0.RVBAR=0x04002000                         \
-C cluster0.cpu1.RVBAR=0x04002000                         \
-C cluster0.cpu2.RVBAR=0x04002000                         \
-C cluster0.cpu3.RVBAR=0x04002000                         \
-C cluster1.cpu0.RVBAR=0x04002000                         \
-C cluster1.cpu1.RVBAR=0x04002000                         \
-C cluster1.cpu2.RVBAR=0x04002000                         \
-C cluster1.cpu3.RVBAR=0x04002000                         \
--data cluster0.cpu0="<path-to>/<bl32-binary>"@0x04002000    \
--data cluster0.cpu0="<path-to>/<bl33-binary>"@0x88000000    \
--data cluster0.cpu0="<path-to>/<fdt>"@0x82000000            \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000  \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Note: The load address of `<bl32-binary>` depends on the value `BL32_BASE`. It should match the address programmed into the RVBAR register as well.

Running on the Cortex-A57-A53 Base FVP with reset to BL31 entrypoint

The following `FVP_Base_Cortex-A57x4-A53x4` model parameters should be used to boot Linux with 8 CPUs using the AArch64 build of TF-A.

```
<path-to>/FVP_Base_Cortex-A57x4-A53x4                           \
-C pctl.startup=0.0.0.0                                         \
-C bp.secure_memory=1                                           \
-C bp.tzc_400.diagnostics=1                                     \
-C cache_state_modelled=1                                       \
-C cluster0.cpu0.RVBARADDR=0x04010000                           \
-C cluster0.cpu1.RVBARADDR=0x04010000                           \
-C cluster0.cpu2.RVBARADDR=0x04010000                           \
-C cluster0.cpu3.RVBARADDR=0x04010000                           \
-C cluster1.cpu0.RVBARADDR=0x04010000                           \
-C cluster1.cpu1.RVBARADDR=0x04010000                           \
-C cluster1.cpu2.RVBARADDR=0x04010000                           \
-C cluster1.cpu3.RVBARADDR=0x04010000                           \
--data cluster0.cpu0="<path-to>/<bl31-binary>"@0x04010000       \
--data cluster0.cpu0="<path-to>/<bl32-binary>"@0xff000000       \
--data cluster0.cpu0="<path-to>/<bl33-binary>"@0x88000000       \
--data cluster0.cpu0="<path-to>/<fdt>"@0x82000000               \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000     \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

Running on the Cortex-A32 Base FVP (AArch32) with reset to SP_MIN entrypoint

The following `FVP_Base_Cortex-A32x4` model parameters should be used to boot Linux with 4 CPUs using the AArch32 build of TF-A.

```
<path-to>/FVP_Base_Cortex-A32x4                                \
-C pctl.startup=0.0.0.0                                        \
-C bp.secure_memory=1                                          \
-C bp.tzc_400.diagnostics=1                                    \
-C cache_state_modelled=1                                      \
-C cluster0.cpu0.RVBARADDR=0x04002000                          \
-C cluster0.cpu1.RVBARADDR=0x04002000                          \
-C cluster0.cpu2.RVBARADDR=0x04002000                          \
-C cluster0.cpu3.RVBARADDR=0x04002000                          \
--data cluster0.cpu0="<path-to>/<bl32-binary>"@0x04002000      \
--data cluster0.cpu0="<path-to>/<bl33-binary>"@0x88000000      \
--data cluster0.cpu0="<path-to>/<fdt>"@0x82000000              \
--data cluster0.cpu0="<path-to>/<kernel-binary>"@0x80080000 \
--data cluster0.cpu0="<path-to>/<ramdisk>"@0x84000000
```

6.2.3 Arm Versatile Express

Versatile Express (VE) family development platform provides an ultra fast environment for prototyping Armv7 System-on-Chip designs. VE Fixed Virtual Platforms (FVP) are simulations of Versatile Express boards. The platform in Trusted Firmware-A has been verified with Arm Cortex-A5 and Cortex-A7 VE FVP's. This platform is tested on and only expected to work with single core models.

Boot Sequence

BL1 –> BL2 –> BL32(sp_min) –> BL33(u-boot) –> Linux kernel

How to build

Code Locations

- U-boot
- Trusted Firmware-A

Build Procedure

- Obtain arm toolchain. The software stack has been verified with linaro 6.2 arm-linux-gnueabihf. Set the CROSS_COMPILE environment variable to point to the toolchain folder.
- Fetch and build u-boot. Make the .config file using the command:

```
make ARCH=arm vexpress_aemv8a_aarch32_config
```

  Make the u-boot binary for Cortex-A5 using the command:

```
make ARCH=arm SUPPORT_ARCH_TIMER=no
```

  Make the u-boot binary for Cortex-A7 using the command:

```
make ARCH=arm
```

- Build TF-A:

  The make command for Cortex-A5 is:

```
make PLAT=fvp_ve ARCH=aarch32 ARM_ARCH_MAJOR=7 ARM_CORTEX_A5=yes \
AARCH32_SP=sp_min FVP_HW_CONFIG_DTS=fdts/fvp-ve-Cortex-A5x1.dts \
ARM_XLAT_TABLES_LIB_V1=1 BL33=<path_to_u-boot.bin> all fip
```

  The make command for Cortex-A7 is:

```
make PLAT=fvp_ve ARCH=aarch32 ARM_ARCH_MAJOR=7 ARM_CORTEX_A7=yes \
AARCH32_SP=sp_min FVP_HW_CONFIG_DTS=fdts/fvp-ve-Cortex-A7x1.dts  \
BL33=<path_to_u-boot.bin> all fip
```

Run Procedure

The following model parameters should be used to boot Linux using the build of Trusted Firmware-A made using the above make commands:

```
./<path_to_model> <path_to_bl1.elf> \
      -C motherboard.flashloader1.fname=<path_to_fip.bin> \
      --data cluster.cpu0=<path_to_zImage>@0x80080000  \
      --data cluster.cpu0=<path_to_ramdisk>@0x84000000
```

6.2.4 TC0 Total Compute Platform

Some of the features of TC0 platform referenced in TF-A include:

- A System Control Processor to abstract power and system management tasks away from application processors. The RAM firmware for SCP is included in the TF-A FIP and is loaded by AP BL2 from FIP in flash to SRAM for copying by SCP (SCP has access to AP SRAM).
- GICv4
- Trusted Board Boot
- SCMI
- MHUv2

Boot Sequence

The execution begins from SCP_BL1. SCP_BL1 powers up the AP which starts executing AP_BL1 and then executes AP_BL2 which loads the SCP_BL2 from FIP to SRAM. The SCP has access to AP SRAM. The address and size of SCP_BL2 is communicated to SCP using SDS. SCP copies SCP_BL2 from SRAM to its own RAM and starts executing it. The AP then continues executing the rest of TF-A stages including BL31 runtime stage and hands off executing to Non-secure world (u-boot).

Build Procedure (TF-A only)

- Obtain arm toolchain. Set the CROSS_COMPILE environment variable to point to the toolchain folder.
- Build TF-A:

  ```
  make PLAT=tc0 BL33=<path_to_uboot.bin> \
  SCP_BL2=<path_to_scp_ramfw.bin>  all fip
  ```

  Enable TBBR by adding the following options to the make command:

  ```
  MBEDTLS_DIR=<path_to_mbedtls_directory>  \
  TRUSTED_BOARD_BOOT=1 \
  GENERATE_COT=1 \
  ARM_ROTPK_LOCATION=devel_rsa  \
  ROT_KEY=plat/arm/board/common/rotpk/arm_rotprivk_rsa.pem
  ```

6.2.5 Arm FPGA Platform

This platform supports FPGA images used internally in Arm Ltd., for testing and bringup of new cores. With that focus, peripheral support is minimal: there is no mass storage or display output, for instance. Also this port ignores any power management features of the platform. Some interconnect setup is done internally by the platform, so the TF-A code just needs to setup UART and GIC.

The FPGA platform requires to pass on a DTB for the non-secure payload (mostly Linux), so we let TF-A use information from the DTB for dynamic configuration: the UART and GIC base addresses are read from there.

As a result this port is a fairly generic BL31-only port, which can serve as a template for a minimal new (and possibly DT-based) platform port.

The aim of this port is to support as many FPGA images as possible with a single build. Image specific data must be described in the DTB or should be auto-detected at runtime.

As the number and topology layout of the CPU cores differs significantly across the various images, this is detected at runtime by BL31. The /cpus node in the DT will be added and filled accordingly, as long as it does not exist already.

Platform-specific build options

- `SUPPORT_UNKNOWN_MPID` : Boolean option to allow unknown MPIDR registers. Normally TF-A panics if it encounters a MPID value not matched to its internal list, but for new or experimental cores this creates a lot of churn. With this option, the code will fall back to some basic CPU support code (only architectural system registers, and no errata). Default value of this flag is 1.
- `PRELOADED_BL33_BASE` : Physical address of the BL33 non-secure payload. It must have been loaded into DRAM already, typically this is done by the script that also loads BL31 and the DTB. It defaults to 0x80080000, which is the traditional load address for an arm64 Linux kernel.
- `FPGA_PRELOADED_DTB_BASE` : Physical address of the flattened device tree blob (DTB). This DT will be used by TF-A for dynamic configuration, so it must describe at least the UART and a GICv3 interrupt controller. The DT gets amended by the code, to potentially add a command line and fill the CPU topology nodes. It will also be passed on to BL33, by putting its address into the x0 register before jumping to the entry point (following the Linux kernel boot protocol). It defaults to 0x80070000, which is 64KB before the BL33 load address.
- `FPGA_PRELOADED_CMD_LINE` : Physical address of the command line to put into the devicetree blob. Due to the lack of a proper bootloader, a command line can be put somewhere into memory, so that BL31 will detect it and copy it into the DTB passed on to BL33. To avoid random garbage, there needs to be a "CMD:" signature before the actual command line. Defaults to 0x1000, which is normally in the "ROM" space of the typical FPGA image (which can be written by the FPGA payload uploader, but is read-only to the CPU). The FPGA payload tool should be given a text file containing the desired command line, prefixed by the "CMD:" signature.

Building the TF-A image

```
make PLAT=arm_fgpa DEBUG=1
```

This will use the default load addresses as described above. When those addresses need to differ for a certain setup, they can be passed on the make command line:

```
make PLAT=arm_fgpa DEBUG=1 PRELOADED_BL33_BASE=0x80200000 FPGA_PRELOADED_DTB_
↪BASE=0x80180000 bl31
```

Running the TF-A image

After building TF-A, the actual TF-A code will be located in `bl31.bin` in the build directory. Additionally there is a `bl31.axf` ELF file, which contains BL31, as well as some simple ROM trampoline code (required by the Arm FPGA boot flow) and a generic DTB to support most of the FPGA images. This can be simply handed over to the FPGA payload uploader, which will take care of loading the components at their respective load addresses. In addition to this file you need at least a BL33 payload (typically a Linux kernel image), optionally a Linux initrd image file and possibly a command line:

```
fpga-run ... -m bl31.axf -l auto -m Image -l 0x80080000 -m initrd.gz -l␣
↪0x84000000 -m cmdline.txt -l 0x1000
```

6.2.6 Arm Development Platform Build Options

Arm Platform Build Options

- `ARM_BL31_IN_DRAM`: Boolean option to select loading of BL31 in TZC secured DRAM. By default, BL31 is in the secure SRAM. Set this flag to 1 to load BL31 in TZC secured DRAM. If TSP is present, then setting this option also sets the TSP location to DRAM and ignores the `ARM_TSP_RAM_LOCATION` build flag.
- `ARM_CONFIG_CNTACR`: boolean option to unlock access to the `CNTBase<N>` frame registers by setting the `CNTCTLBase.CNTACR<N>` register bits. The frame number `<N>` is defined by `PLAT_ARM_NSTIMER_FRAME_ID`, which should match the frame used by the Non-Secure image (normally the Linux kernel). Default is true (access to the frame is allowed).
- `ARM_DISABLE_TRUSTED_WDOG`: boolean option to disable the Trusted Watchdog. By default, Arm platforms use a watchdog to trigger a system reset in case an error is encountered during the boot process (for example, when an image could not be loaded or authenticated). The watchdog is enabled in the early platform setup hook at BL1 and disabled in the BL1 prepare exit hook. The Trusted Watchdog may be disabled at build time for testing or development purposes.
- `ARM_LINUX_KERNEL_AS_BL33`: The Linux kernel expects registers x0-x3 to have specific values at boot. This boolean option allows the Trusted Firmware to have a Linux kernel image as BL33 by preparing the registers to these values before jumping to BL33. This option defaults to 0 (disabled). For AArch64 `RESET_TO_BL31` and for AArch32 `RESET_TO_SP_MIN` must be 1 when using it. If this option is set to 1, `ARM_PRELOADED_DTB_BASE` must be set to the location of a device tree blob (DTB) already loaded in memory. The Linux Image address must be specified using the `PRELOADED_BL33_BASE` option.
- `ARM_PLAT_MT`: This flag determines whether the Arm platform layer has to cater for the multi-threading `MT` bit when accessing MPIDR. When this flag is set, the functions which deal with MPIDR assume that the `MT` bit in MPIDR is set and access the bit-fields in MPIDR accordingly. Default value of this flag is 0. Note that this option is not used on FVP platforms.
- `ARM_RECOM_STATE_ID_ENC`: The PSCI1.0 specification recommends an encoding for the construction of composite state-ID in the power-state parameter. The existing PSCI clients currently do not support this encoding of State-ID yet. Hence this flag is used to configure whether to use the recommended State-ID encoding or not. The default value of this flag is 0, in which case the platform is configured to expect NULL in the State-ID field of power-state parameter.
- `ARM_ROTPK_LOCATION`: used when `TRUSTED_BOARD_BOOT=1`. It specifies the location of the ROTPK hash returned by the function `plat_get_rotpk_info()` for Arm platforms. Depending on the selected option, the proper private key must be specified using the `ROT_KEY` option when building the Trusted Firmware.

This private key will be used by the certificate generation tool to sign the BL2 and Trusted Key certificates. Available options for ARM_ROTPK_LOCATION are:

  - regs : return the ROTPK hash stored in the Trusted root-key storage registers.
  - devel_rsa : return a development public key hash embedded in the BL1 and BL2 binaries. This hash has been obtained from the RSA public key arm_rotpk_rsa.der, located in plat/arm/board/common/rotpk. To use this option, arm_rotprivk_rsa.pem must be specified as ROT_KEY when creating the certificates.
  - devel_ecdsa : return a development public key hash embedded in the BL1 and BL2 binaries. This hash has been obtained from the ECDSA public key arm_rotpk_ecdsa.der, located in plat/arm/board/common/rotpk. To use this option, arm_rotprivk_ecdsa.pem must be specified as ROT_KEY when creating the certificates.

- ARM_ROTPK_HASH: used when ARM_ROTPK_LOCATION=devel_*. Specifies the location of the ROTPK hash. Not expected to be a build option. This defaults to plat/arm/board/common/rotpk/*_sha256.bin depending on the specified algorithm. Providing ROT_KEY enforces generation of the hash from the ROT_KEY and overwrites the default hash file.
- ARM_TSP_RAM_LOCATION: location of the TSP binary. Options:
  - tsram : Trusted SRAM (default option when TBB is not enabled)
  - tdram : Trusted DRAM (if available)
  - dram : Secure region in DRAM (default option when TBB is enabled, configured by the TrustZone controller)
- ARM_XLAT_TABLES_LIB_V1: boolean option to compile TF-A with version 1 of the translation tables library instead of version 2. It is set to 0 by default, which selects version 2.
- ARM_CRYPTOCELL_INTEG : bool option to enable TF-A to invoke Arm® TrustZone® CryptoCell functionality for Trusted Board Boot on capable Arm platforms. If this option is specified, then the path to the CryptoCell SBROM library must be specified via CCSBROM_LIB_PATH flag.
- ARM_SPMC_MANIFEST_DTS : path to an alternate manifest file used as the SPMC Core manifest. Valid when SPD=spmd is selected.

For a better understanding of these options, the Arm development platform memory map is explained in the *Firmware Design*.

Arm CSS Platform-Specific Build Options

- CSS_DETECT_PRE_1_7_0_SCP: Boolean flag to detect SCP version incompatibility. Version 1.7.0 of the SCP firmware made a non-backwards compatible change to the MTL protocol, used for AP/SCP communication. TF-A no longer supports earlier SCP versions. If this option is set to 1 then TF-A will detect if an earlier version is in use. Default is 1.
- CSS_LOAD_SCP_IMAGES: Boolean flag, which when set, adds SCP_BL2 and SCP_BL2U to the FIP and FWU_FIP respectively, and enables them to be loaded during boot. Default is 1.
- CSS_USE_SCMI_SDS_DRIVER: Boolean flag which selects SCMI/SDS drivers instead of SCPI/BOM driver for communicating with the SCP during power management operations and for SCP RAM Firmware transfer. If this option is set to 1, then SCMI/SDS drivers will be used. Default is 0.
- CSS_SGI_CHIP_COUNT: Configures the number of chips on a SGI/RD platform which supports multi-chip operation. If CSS_SGI_CHIP_COUNT is set to any valid value greater than 1, the platform code performs required configuration to support multi-chip operation.

6.2.7 Morello Platform

Morello is an ARMv8-A platform that implements the capability architecture extension. The platform port present at site provides ARMv8-A architecture enablement.

Capability architecture specific changes will be added here

Further information on Morello Platform is available at info

Boot Sequence

The execution begins from SCP_BL1 which loads the SCP_BL2 and starts its execution. SCP_BL2 powers up the AP which starts execution at AP_BL31. The AP then continues executing and hands off execution to Non-secure world (UEFI).

Build Procedure (TF-A only)

- Obtain arm toolchain. Set the CROSS_COMPILE environment variable to point to the toolchain folder.
- Build TF-A:

```
export CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-none-elf-

make PLAT=morello all
```

This chapter holds documentation related to Arm's development platforms, including both software models (FVPs) and hardware development boards such as Juno.

6.3 Amlogic Meson A113D (AXG)

The Amlogic Meson A113D is a SoC with a quad core Arm Cortex-A53 running at ~1.2GHz. It also contains a Cortex-M3 used as SCP.

This port is a minimal implementation of BL31 capable of booting mainline U-Boot and Linux:

- SCPI support.
- Basic PSCI support (CPU_ON, CPU_OFF, SYSTEM_RESET, SYSTEM_OFF). Note that CPU0 can't be turned off, so there is a workaround to hide this from the caller.
- GICv2 driver set up.
- Basic SIP services (read efuse data, enable/disable JTAG).

In order to build it:

```
CROSS_COMPILE=aarch64-none-elf- make DEBUG=1 PLAT=axg [SPD=opteed]
     [AML_USE_ATOS=1 when using ATOS as BL32]
```

This port has been tested on a A113D board. After building it, follow the instructions in the U-Boot repository, replacing the mentioned **bl31.img** by the one built from this port.

6.4 Amlogic Meson S905 (GXBB)

The Amlogic Meson S905 is a SoC with a quad core Arm Cortex-A53 running at 1.5Ghz. It also contains a Cortex-M3 used as SCP.

This port is a minimal implementation of BL31 capable of booting mainline U-Boot and Linux:

- SCPI support.
- Basic PSCI support (CPU_ON, CPU_OFF, SYSTEM_RESET, SYSTEM_OFF). Note that CPU0 can't be turned off, so there is a workaround to hide this from the caller.
- GICv2 driver set up.
- Basic SIP services (read efuse data, enable/disable JTAG).

In order to build it:

```
CROSS_COMPILE=aarch64-linux-gnu- make DEBUG=1 PLAT=gxbb bl31
```

This port has been tested in a ODROID-C2. After building it, follow the instructions in the U-Boot repository, replacing the mentioned **bl31.bin** by the one built from this port.

6.5 Amlogic Meson S905x (GXL)

The Amlogic Meson S905x is a SoC with a quad core Arm Cortex-A53 running at 1.5Ghz. It also contains a Cortex-M3 used as SCP.

This port is a minimal implementation of BL31 capable of booting mainline U-Boot and Linux:

- SCPI support.
- Basic PSCI support (CPU_ON, CPU_OFF, SYSTEM_RESET, SYSTEM_OFF). Note that CPU0 can't be turned off, so there is a workaround to hide this from the caller.
- GICv2 driver set up.
- Basic SIP services (read efuse data, enable/disable JTAG).

In order to build it:

```
CROSS_COMPILE=aarch64-linux-gnu- make DEBUG=1 PLAT=gxl
```

This port has been tested on a Lepotato. After building it, follow the instructions in the gxlimg repository or U-Boot repository, replacing the mentioned **bl31.img** by the one built from this port.

6.6 Amlogic Meson S905X2 (G12A)

The Amlogic Meson S905X2 is a SoC with a quad core Arm Cortex-A53 running at ~1.8GHz. It also contains a Cortex-M3 used as SCP.

This port is a minimal implementation of BL31 capable of booting mainline U-Boot and Linux:

- SCPI support.
- Basic PSCI support (CPU_ON, CPU_OFF, SYSTEM_RESET, SYSTEM_OFF). Note that CPU0 can't be turned off, so there is a workaround to hide this from the caller.
- GICv2 driver set up.
- Basic SIP services (read efuse data, enable/disable JTAG).

In order to build it:

```
CROSS_COMPILE=aarch64-linux-gnu- make DEBUG=1 PLAT=g12a
```

This port has been tested on a SEI510 board. After building it, follow the instructions in the gxlimg repository or U-Boot repository, replacing the mentioned **bl31.img** by the one built from this port.

6.7 HiKey

HiKey is one of 96boards. Hisilicon Kirin6220 processor is installed on HiKey.

More information are listed in link.

6.7.1 How to build

Code Locations

- Trusted Firmware-A: link
- OP-TEE link
- edk2: link
- OpenPlatformPkg: link
- l-loader: link
- atf-fastboot: link

Build Procedure

- Fetch all the above repositories into local host. Make all the repositories in the same ${BUILD_PATH}.

```
git clone https://github.com/ARM-software/arm-trusted-firmware -b integration
git clone https://github.com/OP-TEE/optee_os
git clone https://github.com/96boards-hikey/edk2 -b testing/hikey960_v2.5
git clone https://github.com/96boards-hikey/OpenPlatformPkg -b testing/
↪hikey960_v1.3.4
git clone https://github.com/96boards-hikey/l-loader -b testing/hikey960_v1.2
git clone https://github.com/96boards-hikey/atf-fastboot
```

- Create the symbol link to OpenPlatformPkg in edk2.

```
$cd ${BUILD_PATH}/edk2
$ln -sf ../OpenPlatformPkg
```

- Prepare AARCH64 && AARCH32 toolchain. Prepare python.
- If your hikey hardware is built by CircuitCo, update *OpenPlatformPkg/Platforms/Hisilicon/HiKey/HiKey.dsc* first. *(optional)* console on hikey.**

```
DEFINE SERIAL_BASE=0xF8015000
```

  If your hikey hardware is built by LeMaker, nothing to do.

- Build it as debug mode. Create your own build script file or you could refer to **build_uefi.sh** in l-loader git repository.

```
cd {BUILD_PATH}/arm-trusted-firmware
sh ../l-loader/build_uefi.sh hikey
```

- Generate l-loader.bin and partition table for aosp. The eMMC capacity is either 8GB or 4GB. Just change "aosp-8g" to "linux-8g" for debian.

```
cd ${BUILD_PATH}/l-loader
ln -sf ${EDK2_OUTPUT_DIR}/FV/bl1.bin
ln -sf ${EDK2_OUTPUT_DIR}/FV/bl2.bin
ln -sf ${BUILD_PATH}/atf-fastboot/build/hikey/${FASTBOOT_BUILD_OPTION}/bl1.bin␣
↪fastboot.bin
make hikey PTABLE_LST=aosp-8g
```

6.7.2 Setup Console

- Install ser2net. Use telnet as the console since UEFI fails to display Boot Manager GUI in minicom. **If you don't need Boot Manager GUI, just ignore this section.**

```
$sudo apt-get install ser2net
```

- Configure ser2net.

```
$sudo vi /etc/ser2net.conf
```

  Append one line for serial-over-USB in below. *#ser2net.conf*

```
2004:telnet:0:/dev/ttyUSB0:115200 8DATABITS NONE 1STOPBIT banner
```

- Start ser2net

```
$sudo killall ser2net
$sudo ser2net -u
```

- Open the console.

```
$telnet localhost 2004
```

  And you could open the console remotely, too.

6.7.3 Flash images in recovery mode

- Make sure Pin3-Pin4 on J15 are connected for recovery mode. Then power on HiKey.
- Remove the modemmanager package. This package may cause the idt tool failure.

```
$sudo apt-get purge modemmanager
```

- Run the command to download recovery.bin into HiKey.

```
$sudo python hisi-idt.py -d /dev/ttyUSB1 --img1 recovery.bin
```

- Update images. All aosp or debian images could be fetched from link.

```
$sudo fastboot flash ptable prm_ptable.img
$sudo fastboot flash loader l-loader.bin
$sudo fastboot flash fastboot fip.bin
$sudo fastboot flash boot boot.img
$sudo fastboot flash cache cache.img
$sudo fastboot flash system system.img
$sudo  fastboot flash userdata userdata.img
```

6.7.4 Boot UEFI in normal mode

- Make sure Pin3-Pin4 on J15 are open for normal boot mode. Then power on HiKey.
- Reference link

6.8 HiKey960

HiKey960 is one of 96boards. Hisilicon Hi3660 processor is installed on HiKey960.

More information are listed in link.

6.8.1 How to build

Code Locations

- Trusted Firmware-A: link
- OP-TEE: link
- edk2: link
- OpenPlatformPkg: link
- l-loader: link

Build Procedure

- Fetch all the above 5 repositories into local host. Make all the repositories in the same ${BUILD_PATH}.

```
git clone https://github.com/ARM-software/arm-trusted-firmware -b integration
git clone https://github.com/OP-TEE/optee_os
git clone https://github.com/96boards-hikey/edk2 -b testing/hikey960_v2.5
git clone https://github.com/96boards-hikey/OpenPlatformPkg -b testing/
↪hikey960_v1.3.4
git clone https://github.com/96boards-hikey/l-loader -b testing/hikey960_v1.2
```

- Create the symbol link to OpenPlatformPkg in edk2.

```
$cd ${BUILD_PATH}/edk2
$ln -sf ../OpenPlatformPkg
```

- Prepare AARCH64 toolchain.
- If your hikey960 hardware is v1, update *OpenPlatformPkg/Platforms/Hisilicon/HiKey960/HiKey960.dsc* first. *(optional)*

```
DEFINE SERIAL_BASE=0xFDF05000
```

  If your hikey960 hardware is v2 or newer, nothing to do.

- Build it as debug mode. Create script file for build.

```
cd {BUILD_PATH}/arm-trusted-firmware
sh ../l-loader/build_uefi.sh hikey960
```

- Generate l-loader.bin and partition table. *Make sure that you're using the sgdisk in the l-loader directory.*

```
cd ${BUILD_PATH}/l-loader
ln -sf ${EDK2_OUTPUT_DIR}/FV/bl1.bin
ln -sf ${EDK2_OUTPUT_DIR}/FV/bl2.bin
ln -sf ${EDK2_OUTPUT_DIR}/FV/fip.bin
ln -sf ${EDK2_OUTPUT_DIR}/FV/BL33_AP_UEFI.fd
make hikey960
```

6.8.2 Setup Console

- Install ser2net. Use telnet as the console since UEFI will output window that fails to display in minicom.

```
$sudo apt-get install ser2net
```

- Configure ser2net.

```
$sudo vi /etc/ser2net.conf
```

  Append one line for serial-over-USB in *#ser2net.conf*

```
2004:telnet:0:/dev/ttyUSB0:115200 8DATABITS NONE 1STOPBIT banner
```

- Start ser2net

```
$sudo killall ser2net
$sudo ser2net -u
```

- Open the console.

```
$telnet localhost 2004
```

And you could open the console remotely, too.

6.8.3 Boot UEFI in recovery mode

- Fetch that are used in recovery mode. The code location is in below. link
- Prepare recovery binary.

```
$cd tools-images-hikey960
$ln -sf ${BUILD_PATH}/l-loader/l-loader.bin
$ln -sf ${BUILD_PATH}/l-loader/fip.bin
$ln -sf ${BUILD_PATH}/l-loader/recovery.bin
```

- Prepare config file.

```
$vi config
# The content of config file
./sec_usb_xloader.img 0x00020000
./sec_uce_boot.img 0x6A908000
./recovery.bin 0x1AC00000
```

- Remove the modemmanager package. This package may causes hikey_idt tool failure.

```
$sudo apt-get purge modemmanager
```

- Run the command to download recovery.bin into HiKey960.

```
$sudo ./hikey_idt -c config -p /dev/ttyUSB1
```

- UEFI running in recovery mode. When prompt '.' is displayed on console, press hotkey 'f' in keyboard. Then Android fastboot app is running. The timeout of prompt '.' is 10 seconds.
- Update images.

```
$sudo fastboot flash ptable prm_ptable.img
$sudo fastboot flash xloader sec_xloader.img
$sudo fastboot flash fastboot l-loader.bin
$sudo fastboot flash fip fip.bin
$sudo fastboot flash boot boot.img
$sudo fastboot flash cache cache.img
$sudo fastboot flash system system.img
$sudo fastboot flash userdata userdata.img
```

- Notice: UEFI could also boot kernel in recovery mode, but BL31 isn't loaded in recovery mode.

6.8.4 Boot UEFI in normal mode

- Make sure “Boot Mode” switch is OFF for normal boot mode. Then power on HiKey960.
- Reference link

6.9 Intel Agilex SoCFPGA

Agilex SoCFPGA is a FPGA with integrated quad-core 64-bit Arm Cortex A53 processor.

Upon boot, Boot ROM loads bl2 into OCRAM. Bl2 subsequently initializes the hardware, then loads bl31 and bl33 (UEFI) into DDR and boots to bl33.

```
Boot ROM --> Trusted Firmware-A --> UEFI
```

6.9.1 How to build

Code Locations

- Trusted Firmware-A: link
- UEFI (to be updated with new upstreamed UEFI): link

Build Procedure

- Fetch all the above 2 repositories into local host. Make all the repositories in the same ${BUILD_PATH}.
- Prepare the AARCH64 toolchain.
- Build UEFI using Agilex platform as configuration This will be updated to use an updated UEFI using the latest EDK2 source

```
make CROSS_COMPILE=aarch64-linux-gnu- device=agx
```

- Build atf providing the previously generated UEFI as the BL33 image

```
make CROSS_COMPILE=aarch64-linux-gnu- bl2 fip PLAT=agilex
BL33=PEI.ROM
```

Install Procedure

- dd fip.bin to a A2 partition on the MMC drive to be booted in Agilex board.
- Generate a SOF containing bl2

```
aarch64-linux-gnu-objcopy -I binary -O ihex --change-addresses 0xffe00000 bl2.bin bl2.
↪hex
quartus_cpf --bootloader bl2.hex <quartus_generated_sof> <output_sof_with_bl2>
```

- Configure SOF to board

```
nios2-configure-sof <output_sof_with_bl2>
```

6.9.2 Boot trace

```
INFO:    DDR: DRAM calibration success.
INFO:    ECC is disabled.
NOTICE:  BL2: v2.1(debug)
NOTICE:  BL2: Built
INFO:    BL2: Doing platform setup
NOTICE:  BL2: Booting BL31
INFO:    Entry point address = 0xffe1c000
INFO:    SPSR = 0x3cd
NOTICE:  BL31: v2.1(debug)
NOTICE:  BL31: Built
INFO:    ARM GICv2 driver initialized
INFO:    BL31: Initializing runtime services
WARNING: BL31: cortex_a53
INFO:    BL31: Preparing for EL3 exit to normal world
INFO:    Entry point address = 0x50000
INFO:    SPSR = 0x3c9
```

6.10 Intel Stratix 10 SoCFPGA

Stratix 10 SoCFPGA is a FPGA with integrated quad-core 64-bit Arm Cortex A53 processor.

Upon boot, Boot ROM loads bl2 into OCRAM. Bl2 subsequently initializes the hardware, then loads bl31 and bl33 (UEFI) into DDR and boots to bl33.

```
Boot ROM --> Trusted Firmware-A --> UEFI
```

6.10.1 How to build

Code Locations

- Trusted Firmware-A: link
- UEFI (to be updated with new upstreamed UEFI): link

Build Procedure

- Fetch all the above 2 repositories into local host. Make all the repositories in the same ${BUILD_PATH}.
- Prepare the AARCH64 toolchain.
- Build UEFI using Stratix 10 platform as configuration This will be updated to use an updated UEFI using the latest EDK2 source

```
make CROSS_COMPILE=aarch64-linux-gnu- device=s10
```

- Build atf providing the previously generated UEFI as the BL33 image

```
make CROSS_COMPILE=aarch64-linux-gnu- bl2 fip PLAT=stratix10
BL33=PEI.ROM
```

Install Procedure

- dd fip.bin to a A2 partition on the MMC drive to be booted in Stratix 10 board.
- Generate a SOF containing bl2

```
aarch64-linux-gnu-objcopy -I binary -O ihex --change-addresses 0xffe00000 bl2.bin bl2.
↪hex
quartus_cpf --bootloader bl2.hex <quartus_generated_sof> <output_sof_with_bl2>
```

- Configure SOF to board

```
nios2-configure-sof <output_sof_with_bl2>
```

6.10.2 Boot trace

```
INFO:    DDR: DRAM calibration success.
INFO:    ECC is disabled.
INFO:    Init HPS NOC's DDR Scheduler.
NOTICE:  BL2: v2.0(debug):v2.0-809-g7f8474a-dirty
NOTICE:  BL2: Built : 17:38:19, Feb 18 2019
INFO:    BL2: Doing platform setup
INFO:    BL2: Loading image id 3
INFO:    Loading image id=3 at address 0xffe1c000
INFO:    Image id=3 loaded: 0xffe1c000 - 0xffe24034
INFO:    BL2: Loading image id 5
INFO:    Loading image id=5 at address 0x50000
INFO:    Image id=5 loaded: 0x50000 - 0x550000
NOTICE:  BL2: Booting BL31
INFO:    Entry point address = 0xffe1c000
INFO:    SPSR = 0x3cd
NOTICE:  BL31: v2.0(debug):v2.0-810-g788c436-dirty
NOTICE:  BL31: Built : 15:17:16, Feb 20 2019
INFO:    ARM GICv2 driver initialized
INFO:    BL31: Initializing runtime services
WARNING: BL31: cortex_a53: CPU workaround for 855873 was missing!
INFO:    BL31: Preparing for EL3 exit to normal world
INFO:    Entry point address = 0x50000
INFO:    SPSR = 0x3c9
UEFI firmware (version 1.0 built at 11:26:18 on Nov  7 2018)
```

6.11 Marvell

6.11.1 TF-A Build Instructions for Marvell Platforms

This section describes how to compile the Trusted Firmware-A (TF-A) project for Marvell's platforms.

Build Instructions

(1) Set the cross compiler

```
> export CROSS_COMPILE=/path/to/toolchain/aarch64-linux-gnu-
```

(2) Set path for FIP images:

Set U-Boot image path (relatively to TF-A root or absolute path)

```
> export BL33=path/to/u-boot.bin
```

For example: if U-Boot project (and its images) is located at `~/project/u-boot`, BL33 should be `~/project/u-boot/u-boot.bin`

Note: *u-boot.bin* should be used and not *u-boot-spl.bin*

Set MSS/SCP image path (mandatory only for A7K/8K/CN913x)

```
> export SCP_BL2=path/to/mrvl_scp_bl2*.img
```

(3) Armada-37x0 build requires WTP tools installation.

See below in the section "Tools and external components installation". Install ARM 32-bit cross compiler, which is required for building WTMI image for CM3

```
> sudo apt-get install gcc-arm-linux-gnueabi
```

(4) Clean previous build residuals (if any)

```
> make distclean
```

(5) Build TF-A

There are several build options:

- DEBUG

  Default is without debug information (=0). in order to enable it use `DEBUG=1`. Must be disabled when building UART recovery images due to current console driver implementation that is not compatible with Xmodem protocol used for boot image download.

- LOG_LEVEL

  Defines the level of logging which will be purged to the default output port.

  LOG_LEVEL_NONE 0 LOG_LEVEL_ERROR 10 LOG_LEVEL_NOTICE 20 LOG_LEVEL_WARNING 30 LOG_LEVEL_INFO 40 LOG_LEVEL_VERBOSE 50

- USE_COHERENT_MEM

  This flag determines whether to include the coherent memory region in the BL memory map or not.

- LLC_ENABLE

  Flag defining the LLC (L3) cache state. The cache is enabled by default (`LLC_ENABLE=1`).

- LLC_SRAM

Flag enabling the LLC (L3) cache SRAM support. The LLC SRAM is activated and used by Trusted OS (OP-TEE OS, BL32). The TF-A only prepares CCU address translation windows for SRAM address range at BL31 execution stage with window target set to DRAM-0. When Trusted OS activates LLC SRAM, the CCU window target is changed to SRAM. There is no reason to enable this feature if OP-TEE OS built with CFG_WITH_PAGER=n. Only set LLC_SRAM=1 if OP-TEE OS is built with CFG_WITH_PAGER=y.

- MARVELL_SECURE_BOOT

  Build trusted(=1)/non trusted(=0) image, default is non trusted.

- BLE_PATH

  Points to BLE (Binary ROM extension) sources folder. Only required for A7K/8K/CN913x builds. The parameter is optional, its default value is `plat/marvell/armada/a8k/common/ble`.

- MV_DDR_PATH

  For A7K/8K/CN913x, use this parameter to point to mv_ddr driver sources to allow BLE build. For A37x0, it is used for ddr_tool build.

  Usage example: MV_DDR_PATH=path/to/mv_ddr

  The parameter is optional for A7K/8K/CN913x, when this parameter is not set, the mv_ddr sources are expected to be located at: drivers/marvell/mv_ddr. However, the parameter is necessary for A37x0.

  For the mv_ddr source location, check the section "Tools and external components installation"

- CP_NUM

  Total amount of CPs (South Bridge) connected to AP. When the parameter is omitted, the build uses the default number of CPs, which is a number of embedded CPs inside the package: 1 or 2 depending on the SoC used. The parameter is valid for OcteonTX2 CN913x SoC family (PLAT=t9130), which can have external CPs connected to the MCI ports. Valid values with CP_NUM are in a range of 1 to 3.

- DDR_TOPOLOGY

  For Armada37x0 only, the DDR topology map index/name, default is 0.

  Supported Options:

  - DDR3 1CS (0): DB-88F3720-DDR3-Modular (512MB); EspressoBIN (512MB)
  - DDR4 1CS (1): DB-88F3720-DDR4-Modular (512MB)
  - DDR3 2CS (2): EspressoBIN V3-V5 (1GB 2CS)
  - DDR4 2CS (3): DB-88F3720-DDR4-Modular (4GB)
  - DDR3 1CS (4): DB-88F3720-DDR3-Modular (1GB); EspressoBIN V3-V5 (1GB 1CS)
  - DDR4 1CS (5): EspressoBin V7 (1GB)
  - DDR4 2CS (6): EspressoBin V7 (2GB)
  - DDR3 2CS (7): EspressoBin V3-V5 (2GB)
  - CUSTOMER (CUST): Customer board, DDR3 1CS 512MB

- CLOCKSPRESET

  For Armada37x0 only, the clock tree configuration preset including CPU and DDR frequency, default is CPU_800_DDR_800.

  - CPU_600_DDR_600 - CPU at 600 MHz, DDR at 600 MHz

- CPU_800_DDR_800 - CPU at 800 MHz, DDR at 800 MHz
- CPU_1000_DDR_800 - CPU at 1000 MHz, DDR at 800 MHz
- CPU_1200_DDR_750 - CPU at 1200 MHz, DDR at 750 MHz

• BOOTDEV

For Armada37x0 only, the flash boot device, default is `SPINOR`.

Currently, Armada37x0 only supports `SPINOR`, `SPINAND`, `EMMCNORM` and `SATA`:

- SPINOR - SPI NOR flash boot
- SPINAND - SPI NAND flash boot
- EMMCNORM - eMMC Download Mode

  Download boot loader or program code from eMMC flash into CM3 or CA53 Requires full initialization and command sequence
- SATA - SATA device boot

• PARTNUM

For Armada37x0 only, the boot partition number, default is 0.

To boot from eMMC, the value should be aligned with the parameter in U-Boot with name of `CONFIG_SYS_MMC_ENV_PART`, whose value by default is 1. For details about CONFIG_SYS_MMC_ENV_PART, please refer to the U-Boot build instructions.

• WTMI_IMG

For Armada37x0 only, the path of the WTMI image can point to an image which does nothing, an image which supports EFUSE or a customized CM3 firmware binary. The default image is wtmi.bin that built from sources in WTP folder, which is the next option. If the default image is OK, then this option should be skipped.

• WTP

For Armada37x0 only, use this parameter to point to wtptools source code directory, which can be found as a3700_utils.zip in the release. Usage example: `WTP=/path/to/a3700_utils`

For example, in order to build the image in debug mode with log level up to 'notice' level run

```
> make DEBUG=1 USE_COHERENT_MEM=0 LOG_LEVEL=20 PLAT=<MARVELL_PLATFORM>␣
↪all fip
```

And if we want to build a Armada37x0 image in debug mode with log level up to 'notice' level, the image has the preset CPU at 1000 MHz, preset DDR3 at 800 MHz, the DDR topology of DDR4 2CS, the image boot from SPI NOR flash partition 0, and the image is non trusted in WTP, the command line is as following

```
> make DEBUG=1 USE_COHERENT_MEM=0 LOG_LEVEL=20 CLOCKSPRESET=CPU_1000_DDR_
↪800 \
    MARVELL_SECURE_BOOT=0 DDR_TOPOLOGY=3 BOOTDEV=SPINOR PARTNUM=0␣
↪PLAT=a3700 all fip
```

Supported MARVELL_PLATFORM are:

- a3700 (for both A3720 DB and EspressoBin)
- a70x0
- a70x0_amc (for AMC board)

  - a80x0
  - a80x0_mcbin (for MacchiatoBin)
  - t9130 (OcteonTX2 CN913x)

Special Build Flags

- **PLAT_RECOVERY_IMAGE_ENABLE** When set this option to enable secondary recovery function when build atf. In order to build UART recovery image this operation should be disabled for A7K/8K/CN913x because of hardware limitation (boot from secondary image can interrupt UART recovery process). This MACRO definition is set in `plat/marvell/armada/a8k/common/include/platform_def.h` file.
- **DDR32** In order to work in 32bit DDR, instead of the default 64bit ECC DDR, this flag should be set to 1.

For more information about build options, please refer to the *Build Options* document.

Build output

Marvell's TF-A compilation generates 7 files:

- ble.bin - BLe image
- bl1.bin - BL1 image
- bl2.bin - BL2 image
- bl31.bin - BL31 image
- fip.bin - FIP image (contains BL2, BL31 & BL33 (U-Boot) images)
- boot-image.bin - TF-A image (contains BL1 and FIP images)
- flash-image.bin - Image which contains boot-image.bin and SPL image. Should be placed on the boot flash/device.

Tools and external components installation

Armada37x0 Builds require installation of 3 components

(1) ARM cross compiler capable of building images for the service CPU (CM3). This component is usually included in the Linux host packages. On Debian/Ubuntu hosts the default GNU ARM tool chain can be installed using the following command

```
> sudo apt-get install gcc-arm-linux-gnueabi
```

Only if required, the default tool chain prefix `arm-linux-gnueabi-` can be overwritten using the environment variable `CROSS_CM3`. Example for BASH shell

```
> export CROSS_CM3=/opt/arm-cross/bin/arm-linux-gnueabi
```

(2) DDR initialization library sources (mv_ddr) available at the following repository (use the "mv-ddr-devel" branch):

https://github.com/MarvellEmbeddedProcessors/mv-ddr-marvell.git

(3) Armada3700 tools available at the following repository (use the "A3700_utils-armada-18.12-fixed" branch):

https://github.com/MarvellEmbeddedProcessors/A3700-utils-marvell.git

Armada70x0 and Armada80x0 Builds require installation of an additional component

(1) DDR initialization library sources (mv_ddr) available at the following repository (use the "mv-ddr-devel" branch):

https://github.com/MarvellEmbeddedProcessors/mv-ddr-marvell.git

6.11.2 TF-A Porting Guide for Marvell Platforms

This section describes how to port TF-A to a customer board, assuming that the SoC being used is already supported in TF-A.

Source Code Structure

- The customer platform specific code shall reside under `plat/marvell/armada/<soc family>/<soc>_cust` (e.g. 'plat/marvell/armada/a8k/a7040_cust').
- The platform name for build purposes is called `<soc>_cust` (e.g. `a7040_cust`).
- The build system will reuse all files from within the soc directory, and take only the porting files from the customer platform directory.

Files that require porting are located at `plat/marvell/armada/<soc family>/<soc>_cust` directory.

Armada-70x0/Armada-80x0 Porting

SoC Physical Address Map (marvell_plat_config.c)

This file describes the SoC physical memory mapping to be used for the CCU, IOWIN, AXI-MBUS and IOB address decode units (Refer to the functional spec for more details).

In most cases, using the default address decode windows should work OK.

In cases where a special physical address map is needed (e.g. Special size for PCIe MEM windows, large memory mapped SPI flash…), then porting of the SoC memory map is required.

Note: For a detailed information on how CCU, IOWIN, AXI-MBUS & IOB work, please refer to the SoC functional spec, and under `docs/plat/marvell/armada/misc/mvebu-[ccu/iob/amb/io-win].rst` files.

boot loader recovery (marvell_plat_config.c)

- Background:

  Boot rom can skip the current image and choose to boot from next position if a specific value (`0xDEADB002`) is returned by the ble main function. This feature is used for boot loader recovery by booting from a valid flash-image saved in next position on flash (e.g. address 2M in SPI flash).

  Supported options to implement the skip request are:

  - GPIO
  - I2C
  - User defined

- Porting:

  Under marvell_plat_config.c, implement struct skip_image that includes specific board parameters.

  Warning: To disable this feature make sure the struct skip_image is not implemented.

- Example:

In A7040-DB specific implementation (`plat/marvell/armada/a8k/a70x0/board/marvell_plat_config.c`), the image skip is implemented using GPIO: mpp 33 (SW5).

Before resetting the board make sure there is a valid image on the next flash address:

-tftp [valid address] flash-image.bin -sf update [valid address] 0x2000000 [size]

Press reset and keep pressing the button connected to the chosen GPIO pin. A skip image request message is printed on the screen and boot rom boots from the saved image at the next position.

DDR Porting (dram_port.c)

This file defines the dram topology and parameters of the target board.

The DDR code is part of the BLE component, which is an extension of ARM Trusted Firmware (TF-A).

The DDR driver called mv_ddr is released separately apart from TF-A sources.

The BLE and consequently, the DDR init code is executed at the early stage of the boot process.

Each supported platform of the TF-A has its own DDR porting file called dram_port.c located at `atf/plat/marvell/armada/a8k/<platform>/board` directory.

Please refer to '<path_to_mv_ddr_sources>/doc/porting_guide.txt' for detailed porting description.

The build target directory is "build/<platform>/release/ble".

Comphy Porting (phy-porting-layer.h or phy-default-porting-layer.h)

- **Background:** Some of the comphy's parameters value depend on the HW connection between the SoC and the PHY. Every board type has specific HW characteristics like wire length. Due to those differences some comphy parameters vary between board types. Therefore each board type can have its own list of values for all relevant comphy parameters. The PHY porting layer specifies which parameters need to be suited and the board designer should provide relevant values.

  The PHY porting layer simplifies updating static values per board type, which are now grouped in one place.

  Note: The parameters for the same type of comphy may vary even for the same board type, it is because the lanes from comphy-x to some PHY may have different HW characteristic than lanes from comphy-y to the same (multiplexed) or other PHY.

- **Porting:** The porting layer for PHY was introduced in TF-A. There is one file `drivers/marvell/comphy/phy-default-porting-layer.h` which contains the defaults. Those default parameters are used only if there is no appropriate phy-porting-layer.h file under: `plat/marvell/armada/<soc family>/<platform>/board/phy-porting-layer.h`. If the phy-porting-layer.h exists, the phy-default-porting-layer.h is not going to be included.

  Warning: Not all comphy types are already reworked to support the PHY porting layer, currently the porting layer is supported for XFI/SFI and SATA comphy types.

  The easiest way to prepare the PHY porting layer for custom board is to copy existing example to a new platform:

  - cp `plat/marvell/armada/a8k/a80x0/board/phy-porting-layer.h` "plat/marvell/armada/<soc family>/<platform>/board/phy-porting-layer.h"
  - adjust relevant parameters or
  - if different comphy index is used for specific feature, move it to proper table entry and then adjust.

  Note: The final table size with comphy parameters can be different, depending on the CP module count for given SoC type.

- **Example:** Example porting layer for armada-8040-db is under: `plat/marvell/armada/a8k/a80x0/board/phy-porting-layer.h`

  Note: If there is no PHY porting layer for new platform (missing phy-porting-layer.h), the default values are used (drivers/marvell/comphy/phy-default-porting-layer.h) and the user is warned:

  Warning: "Using default comphy parameters - it may be required to suit them for your board".

6.11.3 Address decoding flow and address translation units of Marvell Armada 8K SoC family

```
+--------------------------------------------------------------------------------------
↪-------------+
|                                                              +--------------+    +---
↪-----------+ |
|                                                              | Memory      +-----   ␣
↪DRAM CS    | |
|+------------+ +-----------+ +-----------+                    | Controller  |    +---
↪-----------+ |
||  AP DMA    | |           | |           |                    +--------------+        ␣
↪             |
||  SD/eMMC   | | CA72 CPUs | |  AP MSS   |                    +--------------+        ␣
↪             |
||  MCI-0/1   | |           | |           |                    | Memory      |        ␣
↪             |
|+------+-----+ +--+--------+ +--------+--+  +-------------+    | Controller  |     +--
↪-----------+ |
|       |          |                    |     |            +----- Translaton  |    ␣
↪|AP          | |
|       |          |                    |     |            |    +--------------+    ␣
↪|Configuration| |
|       |          |                   +-----+            +-------------------------
↪Space         | |
|       |          | +--------------+        |  CCU        |                      +--
↪-----------+ |
|       |          | | MMU         +----------+  Windows    |   +------------+      +--
↪-----------+ |
|       |          +-| translation |         |  Lookup     +----          +---------- ␣
↪ AP SPI    | |
|       |            +--------------+        |             |   |           |      +--
↪-----------+ |
|       |            +--------------+        |             |   |  IO       |      +--
↪-----------+ |
|       +------------| SMMU        +----------+             |   |  Windows  +---------- ␣
↪AP MCI0/1  | |
|                    | translation |        +-------------+   |  Lookup    |      +--
↪-----------+ |
|                    +----------+---+                          |           |      +--
↪-----------+ |
|             -               |                               |           +---------- ␣
↪ AP STM    | |
|            +------------------                              |           |      +--
↪-----------+ |
| AP          |               |                              +-+----------+          ␣
↪             |
+-----------------------------------------------------------------|----------------------
↪-------------+
+-------------|---------------------------------------------------|----------------------
↪-------------+
| CP          |        +--------------+            +------+------+     +---------
↪-----------+ |
|             |        |              |            |             +-------    SB␣
↪CFG Space     | |
|             |        |   DIOB       |            |            |      +---------
↪-----------+ |
```

(continues on next page)

(continued from previous page)

```
|             |            |   Windows   -----------------  IOB        |      +--------
↪-----------+ |
|             |            |   Control   |               |  Windows   +------| SB␣
↪PCIe-0 - PCIe2 | |
|             |            |             |               |  Lookup    |      +--------
↪-----------+ |
|             |            +------+------+               |            |      +--------
↪-----------+ |
|             |                   |                      |            +-------+    ␣
↪SB NAND      | |
|             |                   |                      +------+------+      +--------
↪-----------+ |
|             |                   |                             |                     ␣
↪             |
|             |                   |                             |                     ␣
↪             |
|   +-------------------+   +-------------+              +------+------+      +--------
↪-----------+ |
|   | Network Engine    |   |             |              |            +--------   SB␣
↪SPI-0/SPI-1    | |
|   | Security Engine   |   | PCIe, MSS   |              |  RUNIT     |      +--------
↪-----------+ |
|   | SATA, USB         |   | DMA         |              |  Windows   |      +--------
↪-----------+ |
|   | SD/eMMC           |   |             |              |  Lookup    +--------    SB␣
↪Device Bus   | |
|   | TDM, I2C          |   |             |              |            |      +--------
↪-----------+ |
|   +-------------------+   +------------+               +-------------+              ␣
↪             |
|                                                                                     ␣
↪             |
+-------------------------------------------------------------------------------------
↪-------------+
```

6.11.4 AMB - AXI MBUS address decoding

AXI to M-bridge decoding unit driver for Marvell Armada 8K and 8K+ SoCs.

The Runit offers a second level of address windows lookup. It is used to map transaction towards the CD BootROM, SPI0, SPI1 and Device bus (NOR).

The Runit contains eight configurable windows. Each window defines a contiguous, address space and the properties associated with that address space.

```
Unit          Bank          ATTR
Device-Bus    DEV_BOOT_CS   0x2F
              DEV_CS0       0x3E
              DEV_CS1       0x3D
              DEV_CS2       0x3B
              DEV_CS3       0x37
SPI-0         SPI_A_CS0     0x1E
              SPI_A_CS1     0x5E
              SPI_A_CS2     0x9E
              SPI_A_CS3     0xDE
```

(continues on next page)

(continued from previous page)

```
            SPI_A_CS4    0x1F
            SPI_A_CS5    0x5F
            SPI_A_CS6    0x9F
            SPI_A_CS7    0xDF
SPI         SPI_B_CS0    0x1A
            SPI_B_CS1    0x5A
            SPI_B_CS2    0x9A
            SPI_B_CS3    0xDA
BOOT_ROM    BOOT_ROM     0x1D
UART        UART         0x01
```

Mandatory functions

- **marvell_get_amb_memory_map** Returns the AMB windows configuration and the number of windows

Mandatory structures

- **amb_memory_map** Array that include the configuration of the windows. Every window/entry is a struct which has 2 parameters:
  - Base address of the window
  - Attribute of the window

Examples

```
struct addr_map_win amb_memory_map[] = {
    {0xf900,        AMB_DEV_CS0_ID},
};
```

6.11.5 Marvell CCU address decoding bindings

CCU configration driver (1st stage address translation) for Marvell Armada 8K and 8K+ SoCs.

The CCU node includes a description of the address decoding configuration.

Mandatory functions

- **marvell_get_ccu_memory_map** Return the CCU windows configuration and the number of windows of the specific AP.

Mandatory structures

- **ccu_memory_map** Array that includes the configuration of the windows. Every window/entry is a struct which has 3 parameters:
  - Base address of the window
  - Size of the window
  - Target-ID of the window

Example

```
struct addr_map_win ccu_memory_map[] = {
        {0x00000000f2000000,     0x00000000e000000,      IO_0_TID}, /* IO window */
};
```

6.11.6 Marvell IO WIN address decoding bindings

IO Window configration driver (2nd stage address translation) for Marvell Armada 8K and 8K+ SoCs.

The IO WIN includes a description of the address decoding configuration.

Transactions that are decoded by CCU windows as IO peripheral, have an additional layer of decoding. This additional address decoding layer defines one of the following targets:

- **0x0** = BootRom
- **0x1** = STM (Serial Trace Macro-cell, a programmer's port into trace stream)
- **0x2** = SPI direct access
- **0x3** = PCIe registers
- **0x4** = MCI Port
- **0x5** = PCIe port

Mandatory functions

- **marvell_get_io_win_memory_map** Returns the IO windows configuration and the number of windows of the specific AP.

Mandatory structures

- **io_win_memory_map** Array that include the configuration of the windows. Every window/entry is a struct which has 3 parameters:
  - Base address of the window
  - Size of the window
  - Target-ID of the window

Example

```
struct addr_map_win io_win_memory_map[] = {
        {0x00000000fe000000,    0x000000001f00000,      PCIE_PORT_TID}, /* PCIe␣
↪window 31Mb for PCIe port*/
        {0x00000000ffe00000,    0x000000000100000,      PCIE_REGS_TID}, /* PCI-REG␣
↪window 64Kb for PCIe-reg*/
        {0x00000000f6000000,    0x000000000100000,      MCIPHY_TID},    /* MCI window␣
↪ 1Mb for PHY-reg*/
};
```

6.11.7 Marvell IOB address decoding bindings

IO bridge configration driver (3rd stage address translation) for Marvell Armada 8K and 8K+ SoCs.

The IOB includes a description of the address decoding configuration.

IOB supports up to n (in CP110 n=24) windows for external memory transaction. When a transaction passes through the IOB, its address is compared to each of the enabled windows. If there is a hit and it passes the security checks, it is advanced to the target port.

Mandatory functions

- **marvell_get_iob_memory_map** Returns the IOB windows configuration and the number of windows

Mandatory structures

- **iob_memory_map** Array that includes the configuration of the windows. Every window/entry is a struct which has 3 parameters:
  - Base address of the window
  - Size of the window
  - Target-ID of the window

Target ID options

- **0x0** = Internal configuration space
- **0x1** = MCI0
- **0x2** = PEX1_X1
- **0x3** = PEX2_X1
- **0x4** = PEX0_X4
- **0x5** = NAND flash
- **0x6** = RUNIT (NOR/SPI/BootRoom)
- **0x7** = MCI1

Example

```
struct addr_map_win iob_memory_map[] = {
        {0x00000000f7000000,    0x0000000001000000,     PEX1_TID}, /* PEX1_X1 window␣
↪*/
        {0x00000000f8000000,    0x0000000001000000,     PEX2_TID}, /* PEX2_X1 window␣
↪*/
        {0x00000000f6000000,    0x0000000001000000,     PEX0_TID}, /* PEX0_X4 window␣
↪*/
        {0x00000000f9000000,    0x0000000001000000,     NAND_TID}  /* NAND window */
};
```

6.12 MediaTek 8183

MediaTek 8183 (MT8183) is a 64-bit ARM SoC introduced by MediaTek in early 2018. The chip incorporates eight cores - four Cortex-A53 little cores and Cortex-A73. Both clusters can operate at up to 2 GHz.

6.12.1 Boot Sequence

```
Boot Rom --> Coreboot --> TF-A BL31 --> Depthcharge --> Linux Kernel
```

6.12.2 How to Build

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=mt8183 DEBUG=1
```

6.13 MediaTek 8192

MediaTek 8192 (MT8192) is a 64-bit ARM SoC introduced by MediaTek in 2020. The chip incorporates eight cores - four Cortex-A55 little cores and Cortex-A76. Cortex-A76 can operate at up to 2.2 GHz. Cortex-A55 can operate at up to 2 GHz.

6.13.1 Boot Sequence

```
Boot Rom --> Coreboot --> TF-A BL31 --> Depthcharge --> Linux Kernel
```

6.13.2 How to Build

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=mt8192 DEBUG=1 COREBOOT=1
```

6.14 NVIDIA Tegra

- **T194**

T194 has eight NVIDIA Carmel CPU cores in a coherent multi-processor configuration. The Carmel cores support the ARM Architecture version 8.2, executing both 64-bit AArch64 code, and 32-bit AArch32 code. The Carmel processors are organized as four dual-core clusters, where each cluster has a dedicated 2 MiB Level-2 unified cache. A high speed coherency fabric connects these processor complexes and allows heterogeneous multi-processing with all eight cores if required.

- **T186**

The NVIDIA® Parker (T186) series system-on-chip (SoC) delivers a heterogeneous multi-processing (HMP) solution designed to optimize performance and efficiency.

T186 has Dual NVIDIA Denver 2 ARM® CPU cores, plus Quad ARM Cortex®-A57 cores, in a coherent multiprocessor configuration. The Denver 2 and Cortex-A57 cores support ARMv8, executing both 64-bit Aarch64 code, and 32-bit Aarch32 code including legacy ARMv7 applications. The Denver 2 processors each have 128 KB Instruction and 64 KB Data Level 1 caches; and have a 2MB shared Level 2 unified cache. The Cortex-A57 processors each have 48 KB Instruction and 32 KB Data Level 1 caches; and also have a 2 MB shared Level 2 unified cache. A high speed coherency fabric connects these two processor complexes and allows heterogeneous multi-processing with all six cores if required.

- **T210**

T210 has Quad Arm® Cortex®-A57 cores in a switched configuration with a companion set of quad Arm Cortex-A53 cores. The Cortex-A57 and A53 cores support Armv8-A, executing both 64-bit Aarch64 code, and 32-bit Aarch32 code including legacy Armv7-A applications. The Cortex-A57 processors each have 48 KB Instruction and 32 KB Data Level 1 caches; and have a 2 MB shared Level 2 unified cache. The Cortex-A53 processors each have 32 KB Instruction and 32 KB Data Level 1 caches; and have a 512 KB shared Level 2 unified cache.

- **T132**

Denver is NVIDIA's own custom-designed, 64-bit, dual-core CPU which is fully Armv8-A architecture compatible. Each of the two Denver cores implements a 7-way superscalar microarchitecture (up to 7 concurrent micro-ops can be executed per clock), and includes a 128KB 4-way L1 instruction cache, a 64KB 4-way L1 data cache, and a 2MB 16-way L2 cache, which services both cores.

Denver implements an innovative process called Dynamic Code Optimization, which optimizes frequently used software routines at runtime into dense, highly tuned microcode-equivalent routines. These are stored in a dedicated, 128MB main-memory-based optimization cache. After being read into the instruction cache, the optimized micro-ops are executed, re-fetched and executed from the instruction cache as long as needed and capacity allows.

Effectively, this reduces the need to re-optimize the software routines. Instead of using hardware to extract the instruction-level parallelism (ILP) inherent in the code, Denver extracts the ILP once via software techniques, and then executes those routines repeatedly, thus amortizing the cost of ILP extraction over the many execution instances.

Denver also features new low latency power-state transitions, in addition to extensive power-gating and dynamic voltage and clock scaling based on workloads.

6.14.1 Directory structure

- plat/nvidia/tegra/common - Common code for all Tegra SoCs
- plat/nvidia/tegra/soc/txxx - Chip specific code

6.14.2 Trusted OS dispatcher

Tegra supports multiple Trusted OS'.

- Trusted Little Kernel (TLK): In order to include the 'tlkd' dispatcher in the image, pass 'SPD=tlkd' on the command line while preparing a bl31 image.
- Trusty: In order to include the 'trusty' dispatcher in the image, pass 'SPD=trusty' on the command line while preparing a bl31 image.

This allows other Trusted OS vendors to use the upstream code and include their dispatchers in the image without changing any makefiles.

These are the supported Trusted OS' by Tegra platforms.

- Tegra132: TLK
- Tegra210: TLK and Trusty
- Tegra186: Trusty
- Tegra194: Trusty

6.14.3 Scatter files

Tegra platforms currently support scatter files and ld.S scripts. The scatter files help support ARMLINK linker to generate BL31 binaries. For now, there exists a common scatter file, plat/nvidia/tegra/scat/bl31.scat, for all Tegra SoCs. The *LINKER* build variable needs to point to the ARMLINK binary for the scatter file to be used. Tegra platforms have verified BL31 image generation with ARMCLANG (compilation) and ARMLINK (linking) for the Tegra186 platforms.

6.14.4 Preparing the BL31 image to run on Tegra SoCs

```
CROSS_COMPILE=<path-to-aarch64-gcc>/bin/aarch64-none-elf- make PLAT=tegra \
TARGET_SOC=<target-soc e.g. t194|t186|t210|t132> SPD=<dispatcher e.g. trusty|tlkd>
bl31
```

Platforms wanting to use different TZDRAM_BASE, can add `TZDRAM_BASE=<value>` to the build command line.

The Tegra platform code expects a pointer to the following platform specific structure via 'x1' register from the BL2 layer which is used by the bl31_early_platform_setup() handler to extract the TZDRAM carveout base and size for loading the Trusted OS and the UART port ID to be used. The Tegra memory controller driver programs this base/size in order to restrict NS accesses.

typedef struct plat_params_from_bl2 { /* TZ memory size / *uint64_t tzdram_size;* / TZ memory base / *uint64_t tzdram_base;* / UART port ID */ int uart_id; /* L2 ECC parity protection disable flag */ int l2_ecc_parity_prot_dis; /* SHMEM base address for storing the boot logs */ uint64_t boot_profiler_shmem_base; } plat_params_from_bl2_t;

6.14.5 Power Management

The PSCI implementation expects each platform to expose the 'power state' parameter to be used during the 'SYSTEM SUSPEND' call. The state-id field is implementation defined on Tegra SoCs and is preferably defined by tegra_def.h.

6.14.6 Tegra configs

- 'tegra_enable_l2_ecc_parity_prot': This flag enables the L2 ECC and Parity Protection bit, for Arm Cortex-A57 CPUs, during CPU boot. This flag will be enabled by Tegrs SoCs during 'Cluster power up' or 'System Suspend' exit.

6.15 NXP i.MX7 WaRP7

The Trusted Firmware-A port for the i.MX7Solo WaRP7 implements BL2 at EL3. The i.MX7S contains a BootROM with a High Assurance Boot (HAB) functionality. This functionality provides a mechanism for establishing a root-of-trust from the reset vector to the command-line in user-space.

6.15.1 Boot Flow

BootROM –> TF-A BL2 –> BL32(OP-TEE) –> BL33(U-Boot) –> Linux

In the WaRP7 port we encapsulate OP-TEE, DTB and U-Boot into a FIP. This FIP is expected and required

6.15.2 Build Instructions

We need to use a file generated by u-boot in order to generate a .imx image the BootROM will boot. It is therefore _required_ to build u-boot before TF-A and furthermore it is _recommended_ to use the mkimage in the u-boot/tools directory to generate the TF-A .imx image.

U-Boot

https://git.linaro.org/landing-teams/working/mbl/u-boot.git

```
git checkout -b rms-atf-optee-uboot linaro-mbl/rms-atf-optee-uboot
make warp7_bl33_defconfig;
make u-boot.imx arch=ARM CROSS_COMPILE=arm-linux-gnueabihf-
```

OP-TEE

https://github.com/OP-TEE/optee_os.git

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- PLATFORM=imx PLATFORM_
↪FLAVOR=mx7swarp7 ARCH=arm CFG_PAGEABLE_ADDR=0 CFG_DT_ADDR=0x83000000 CFG_NS_ENTRY_
↪ADDR=0x87800000
```

TF-A

https://github.com/ARM-software/arm-trusted-firmware.git

The following commands assume that a directory exits in the top-level TFA build directory "fiptool_images". "fiptool_images" contains

- u-boot.bin The binary output from the u-boot instructions above
- tee-header_v2.bin
- tee-pager_v2.bin
- tee-pageable_v2.bin Binary outputs from the previous OPTEE build steps

It is also assumed copy of mbedtls is available on the path path ../mbedtls https://github.com/ARMmbed/mbedtls.git At the time of writing HEAD points to 0592ea772aee48ca1e6d9eb84eca8e143033d973

```
mkdir fiptool_images
cp /path/to/optee/out/arm-plat-imx/core/tee-header_v2.bin fiptool_images
cp /path/to/optee/out/arm-plat-imx/core/tee-pager_v2.bin fiptool_images
cp /path/to/optee/out/arm-plat-imx/core/tee-pageable_v2.bin fiptool_images

make CROSS_COMPILE=${CROSS_COMPILE} PLAT=warp7 ARCH=aarch32 ARM_ARCH_MAJOR=7 \
     ARM_CORTEX_A7=yes AARCH32_SP=optee PLAT_WARP7_UART=1 GENERATE_COT=1 \
     TRUSTED_BOARD_BOOT=1 USE_TBBR_DEFS=1 MBEDTLS_DIR=../mbedtls \
     NEED_BL32=yes BL32=fiptool_images/tee-header_v2.bin \
     BL32_EXTRA1=fiptool_images/tee-pager_v2.bin \
     BL32_EXTRA2=fiptool_images/tee-pageable_v2.bin \
     BL33=fiptool_images/u-boot.bin certificates all

/path/to/u-boot/tools/mkimage -n /path/to/u-boot/u-boot.cfgout -T imximage -e␣
↪0x9df00000 -d ./build/warp7/debug/bl2.bin ./build/warp7/debug/bl2.bin.imx
```

FIP

```
cp /path/to/uboot/u-boot.bin fiptool_images
cp /path/to/linux/arch/boot/dts/imx7s-warp.dtb fiptool_images

tools/cert_create/cert_create -n --rot-key "build/warp7/debug/rot_key.pem" \
           --tfw-nvctr 0 \
           --ntfw-nvctr 0 \
           --trusted-key-cert fiptool_images/trusted-key-cert.key-crt \
           --tb-fw=build/warp7/debug/bl2.bin \
           --tb-fw-cert fiptool_images/trusted-boot-fw.key-crt\
           --tos-fw fiptool_images/tee-header_v2.bin \
           --tos-fw-cert fiptool_images/tee-header_v2.bin.crt \
           --tos-fw-key-cert fiptool_images/tee-header_v2.bin.key-crt \
           --tos-fw-extra1 fiptool_images/tee-pager_v2.bin \
           --tos-fw-extra2 fiptool_images/tee-pageable_v2.bin \
           --nt-fw fiptool_images/u-boot.bin \
           --nt-fw-cert fiptool_images/u-boot.bin.crt \
           --nt-fw-key-cert fiptool_images/u-boot.bin.key-crt \
           --hw-config fiptool_images/imx7s-warp.dtb

tools/fiptool/fiptool create --tos-fw fiptool_images/tee-header_v2.bin \
          --tos-fw-extra1 fiptool_images/tee-pager_v2.bin \
          --tos-fw-extra2 fiptool_images/tee-pageable_v2.bin \
```

(continues on next page)

(continued from previous page)

```
          --nt-fw fiptool_images/u-boot.bin \
          --hw-config fiptool_images/imx7s-warp.dtb \
          --tos-fw-cert fiptool_images/tee-header_v2.bin.crt \
          --tos-fw-key-cert fiptool_images/tee-header_v2.bin.key-crt \
          --nt-fw-cert fiptool_images/u-boot.bin.crt \
          --nt-fw-key-cert fiptool_images/u-boot.bin.key-crt \
          --trusted-key-cert fiptool_images/trusted-key-cert.key-crt \
          --tb-fw-cert fiptool_images/trusted-boot-fw.key-crt warp7.fip
```

6.15.3 Deploy Images

First place the WaRP7 into UMS mode in u-boot this should produce an entry in /dev like /dev/disk/by-id/usb-Linux_UMS_disk_0_WaRP7-0xf42400d3000001d4-0:0

```
=> ums 0 mmc 0
```

Next flash bl2.imx and warp7.fip

bl2.imx is flashed @ 1024 bytes warp7.fip is flash @ 1048576 bytes

```
sudo dd if=bl2.bin.imx of=/dev/disk/by-id/usb-Linux_UMS_disk_0_WaRP7-
↪0xf42400d3000001d4-0\:0 bs=512 seek=2 conv=notrunc
# Offset is 1MB 1048576 => 1048576 / 512 = 2048
sudo dd if=./warp7.fip of=/dev/disk/by-id/usb-Linux_UMS_disk_0_WaRP7-
↪0xf42400d3000001d4-0\:0 bs=512 seek=2048 conv=notrunc
```

Remember to umount the USB device pefore proceeding

```
sudo umount /dev/disk/by-id/usb-Linux_UMS_disk_0_WaRP7-0xf42400d3000001d4-0\:0*
```

6.15.4 Signing BL2

A further step is to sign BL2.

The image_sign.sh and bl2_sign.csf files alluded to blow are available here.

https://github.com/bryanodonoghue/atf-code-signing

It is suggested you use this script plus the example CSF file in order to avoid hard-coding data into your CSF files.

Download both “image_sign.sh” and “bl2_sign.csf” to your arm-trusted-firmware top-level directory.

```
#!/bin/bash
SIGN=image_sign.sh
TEMP=`pwd`/temp
BL2_CSF=bl2_sign.csf
BL2_IMX=bl2.bin.imx
CST_PATH=/path/to/cst-2.3.2
CST_BIN=${CST_PATH}/linux64/cst

#Remove temp
rm -rf ${TEMP}
mkdir ${TEMP}

# Generate IMX header
```

(continues on next page)

(continued from previous page)

```
/path/to/u-boot/tools/mkimage -n u-boot.cfgout.warp7 -T imximage -e 0x9df00000 -d ./
↪build/warp7/debug/bl2.bin ./build/warp7/debug/bl2.bin.imx > ${TEMP}/${BL2_IMX}.log

# Copy required items to $TEMP
cp build/warp7/debug/bl2.bin.imx ${TEMP}
cp ${CST_PATH}/keys/* ${TEMP}
cp ${CST_PATH}/crts/* ${TEMP}
cp ${BL2_CSF} ${TEMP}

# Generate signed BL2 image
./${SIGN} image_sign_mbl_binary ${TEMP} ${BL2_CSF} ${BL2_IMX} ${CST_BIN}

# Copy signed BL2 to top-level directory
cp ${TEMP}/${BL2_IMX}-signed .
cp ${BL2_RECOVER_CSF} ${TEMP}
```

The resulting bl2.bin.imx-signed can replace bl2.bin.imx in the Deploy Images section above, once done.

Suggested flow for verifying.

1. Followed all previous steps above and verify a non-secure ATF boot
2. Down the NXP Code Singing Tool
3. Generate keys
4. Program the fuses on your board
5. Replace bl2.bin.imx with bl2.bin.imx-signed
6. Verify inside u-boot that "hab_status" shows no events
7. Subsequently close your board.

If you have HAB events @ step 6 - do not lock your board.

To get a good over-view of generating keys and programming the fuses on the board read "High Assurance Boot for Dummies" by Boundary Devices.

https://boundarydevices.com/high-assurance-boot-hab-dummies/

6.16 NXP i.MX 8 Series

The i.MX 8 series of applications processors is a feature- and performance-scalable multi-core platform that includes single-, dual-, and quad-core families based on the Arm® Cortex® architecture—including combined Cortex-A72 + Cortex-A53, Cortex-A35, and Cortex-M4 based solutions for advanced graphics, imaging, machine vision, audio, voice, video, and safety-critical applications.

The i.MX8QM is with 2 Cortex-A72 ARM core, 4 Cortex-A53 ARM core and 1 Cortex-M4 system controller.

The i.MX8QX is with 4 Cortex-A35 ARM core and 1 Cortex-M4 system controller.

The System Controller (SC) represents the evolution of centralized control for system-level resources on i.MX8. The heart of the system controller is a Cortex-M4 that executes system controller firmware.

6.16.1 Boot Sequence

Bootrom –> BL31 –> BL33(u-boot) –> Linux kernel

6.16.2 How to build

Build Procedure

- Prepare AARCH64 toolchain.
- Build System Controller Firmware and u-boot firstly, and get binary images: scfw_tcm.bin and u-boot.bin
- Build TF-A

  Build bl31:

  ```
  CROSS_COMPILE=aarch64-linux-gnu- make PLAT=<Target_SoC> bl31
  ```

  Target_SoC should be "imx8qm" for i.MX8QM SoC. Target_SoC should be "imx8qx" for i.MX8QX SoC.

Deploy TF-A Images

TF-A binary(bl31.bin), scfw_tcm.bin and u-boot.bin are combined together to generate a binary file called flash.bin, the imx-mkimage tool is used to generate flash.bin, and flash.bin needs to be flashed into SD card with certain offset for BOOT ROM. The system controller firmware, u-boot and imx-mkimage will be upstreamed soon, this doc will be updated once they are ready, and the link will be posted.

6.17 NXP i.MX 8M Series

The i.MX 8M family of applications processors based on Arm Corte-A53 and Cortex-M4 cores provide high-performance computing, power efficiency, enhanced system reliability and embedded security needed to drive the growth of fast-growing edge node computing, streaming multimedia, and machine learning applications.

6.17.1 Boot Sequence

Bootrom –> SPL –> BL31 –> BL33(u-boot) –> Linux kernel

6.17.2 How to build

Build Procedure

- Prepare AARCH64 toolchain.
- Build spl and u-boot firstly, and get binary images: u-boot-spl.bin, u-boot-nodtb.bin and dtb for the target board.
- Build TF-A

  Build bl31:

  ```
  CROSS_COMPILE=aarch64-linux-gnu- make PLAT=<Target_SoC> bl31
  ```

Target_SoC should be "imx8mq" for i.MX8MQ SoC. Target_SoC should be "imx8mm" for i.MX8MM SoC. Target_SoC should be "imx8mn" for i.MX8MN SoC. Target_SoC should be "imx8mp" for i.MX8MP SoC.

Deploy TF-A Images

TF-A binary(bl31.bin), u-boot-spl.bin u-boot-nodtb.bin and dtb are combined together to generate a binary file called flash.bin, the imx-mkimage tool is used to generate flash.bin, and flash.bin needs to be flashed into SD card with certain offset for BOOT ROM. the u-boot and imx-mkimage will be upstreamed soon, this doc will be updated once they are ready, and the link will be posted.

6.18 NXP QorIQ® LS1043A

The QorIQ® LS1043A processor is NXP's first quad-core, 64-bit Arm®-based processor for embedded networking. The LS1023A (two core version) and the LS1043A (four core version) deliver greater than 10 Gbps of performance in a flexible I/O package supporting fanless designs. This SoC is a purpose-built solution for small-form-factor networking and industrial applications with BOM optimizations for economic low layer PCB, lower cost power supply and single clock design. The new 0.9V versions of the LS1043A and LS1023A deliver addition power savings for applications such as Wireless LAN and to Power over Ethernet systems.

6.18.1 LS1043ARDB Specification:

Memory subsystem:

- 2GByte DDR4 SDRAM (32bit bus)
- 128 Mbyte NOR flash single-chip memory
- 512 Mbyte NAND flash
- 16 Mbyte high-speed SPI flash
- SD connector to interface with the SD memory card

Ethernet:

- XFI 10G port
- QSGMII with 4x 1G ports
- Two RGMII ports

PCIe:

- PCIe2 (Lanes C) to mini-PCIe slot
- PCIe3 (Lanes D) to PCIe slot

USB 3.0: two super speed USB 3.0 type A ports

UART: supports two UARTs up to 115200 bps for console

More information are listed in ls1043.

6.18.2 Boot Sequence

Bootrom –> TF-A BL1 –> TF-A BL2 –> TF-A BL1 –> TF-A BL31 –> BL32(Tee OS) –> TF-A BL31 –> BL33(u-boot) –> Linux kernel

6.18.3 How to build

Build Procedure

- Prepare AARCH64 toolchain.
- Build u-boot and OPTee firstly, and get binary images: u-boot.bin and tee.bin
- Build TF-A for Nor boot

  Build bl1:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=ls1043 bl1
```

  Build fip:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=ls1043 fip \
BL33=u-boot.bin NEED_BL32=yes BL32=tee.bin SPD=opteed
```

Deploy TF-A Images

- Deploy TF-A images on Nor flash Alt Bank.

```
=> tftp 82000000  bl1.bin
=> pro off all;era 64100000 +$filesize;cp.b 82000000 64100000 $filesize

=> tftp 82000000  fip.bin
=> pro off all;era 64120000 +$filesize;cp.b 82000000 64120000 $filesize
```

  Then change to Alt bank and boot up TF-A:

```
=> cpld reset altbank
```

6.19 Poplar

Poplar is the first development board compliant with the 96Boards Enterprise Edition TV Platform specification.

The board features the Hi3798C V200 with an integrated quad-core 64-bit Arm Cortex A53 processor and high performance Mali T720 GPU, making it capable of running any commercial set-top solution based on Linux or Android.

It supports a premium user experience with up to H.265 HEVC decoding of 4K video at 60 frames per second.

```
SOC Hisilicon Hi3798CV200
CPU Quad-core Arm Cortex-A53 64 bit
DRAM DDR3/3L/4 SDRAM interface, maximum 32-bit data width 2 GB
USB Two USB 2.0 ports One USB 3.0 ports
CONSOLE USB-micro port for console support
ETHERNET 1 GBe Ethernet
```

(continues on next page)

(continued from previous page)

```
PCIE One PCIe 2.0 interfaces
JTAG 8-Pin JTAG
EXPANSION INTERFACE Linaro 96Boards Low Speed Expansion slot
DIMENSION Standard 160×120 mm 96Boards Enterprice Edition form factor
WIFI 802.11AC 2*2 with Bluetooth
CONNECTORS One connector for Smart Card One connector for TSI
```

At the start of the boot sequence, the bootROM executes the so called l-loader binary whose main role is to change the processor state to 64bit mode. This must happen prior to invoking Trusted Firmware-A:

```
l-loader --> Trusted Firmware-A --> u-boot
```

6.19.1 How to build

Code Locations

- Trusted Firmware-A: link
- l-loader: link
- u-boot: link

Build Procedure

- Fetch all the above 3 repositories into local host. Make all the repositories in the same ${BUILD_PATH}.
- Prepare the AARCH64 toolchain.
- Build u-boot using poplar_defconfig

```
make CROSS_COMPILE=aarch64-linux-gnu- poplar_defconfig
make CROSS_COMPILE=aarch64-linux-gnu-
```

- Build atf providing the previously generated u-boot.bin as the BL33 image

```
make CROSS_COMPILE=aarch64-linux-gnu-  all fip SPD=none PLAT=poplar
BL33=u-boot.bin
```

- **Build l-loader (generated the final fastboot.bin)**
  1. copy the atf generated files fip.bin and bl1.bin to l-loader/atf/
  2. export ARM_TRUSTED_FIRMWARE=${ATF_SOURCE_PATH)
  3. make

6.19.2 Install Procedure

- Copy l-loader/fastboot.bin to a FAT partition on a USB pen drive.
- Plug the USB pen drive to any of the USB2 ports
- Power the board while keeping S3 pressed (usb_boot)

The system will boot into a u-boot shell which you can then use to write the working firmware to eMMC.

6.19.3 Boot trace

```
Bootrom start
Boot Media: eMMC
Decrypt auxiliary code ...OK

lsadc voltage min: 000000FE, max: 000000FF, aver: 000000FE, index: 00000000

Entry boot auxiliary code

Auxiliary code - v1.00
DDR code - V1.1.2 20160205
Build: Mar 24 2016 - 17:09:44
Reg Version:  v134
Reg Time:     2016/03/18 09:44:55
Reg Name:     hi3798cv2dmb_hi3798cv200_ddr3_2gbyte_8bitx4_4layers.reg

Boot auxiliary code success
Bootrom success

LOADER:  Switched to aarch64 mode
LOADER:  Entering ARM TRUSTED FIRMWARE
LOADER:  CPU0 executes at 0x000ce000

INFO:    BL1: 0xe1000 - 0xe7000 [size = 24576]
NOTICE:  Booting Trusted Firmware
NOTICE:  BL1: v1.3(debug):v1.3-372-g1ba9c60
NOTICE:  BL1: Built : 17:51:33, Apr 30 2017
INFO:    BL1: RAM 0xe1000 - 0xe7000
INFO:    BL1: Loading BL2
INFO:    Loading image id=1 at address 0xe9000
INFO:    Image id=1 loaded at address 0xe9000, size = 0x5008
NOTICE:  BL1: Booting BL2
INFO:    Entry point address = 0xe9000
INFO:    SPSR = 0x3c5
NOTICE:  BL2: v1.3(debug):v1.3-372-g1ba9c60
NOTICE:  BL2: Built : 17:51:33, Apr 30 2017
INFO:    BL2: Loading BL31
INFO:    Loading image id=3 at address 0x129000
INFO:    Image id=3 loaded at address 0x129000, size = 0x8038
INFO:    BL2: Loading BL33
INFO:    Loading image id=5 at address 0x37000000
INFO:    Image id=5 loaded at address 0x37000000, size = 0x58f17
NOTICE:  BL1: Booting BL31
INFO:    Entry point address = 0x129000
INFO:    SPSR = 0x3cd
INFO:    Boot bl33 from 0x37000000 for 364311 Bytes
```

(continues on next page)

(continued from previous page)

```
NOTICE:  BL31: v1.3(debug):v1.3-372-g1ba9c60
NOTICE:  BL31: Built : 17:51:33, Apr 30 2017
INFO:    BL31: Initializing runtime services
INFO:    BL31: Preparing for EL3 exit to normal world
INFO:    Entry point address = 0x37000000
INFO:    SPSR = 0x3c9


U-Boot 2017.05-rc2-00130-gd2255b0 (Apr 30 2017 - 17:51:28 +0200)poplar

Model: HiSilicon Poplar Development Board
BOARD: Hisilicon HI3798cv200 Poplar
DRAM:  1 GiB
MMC:   Hisilicon DWMMC: 0
In:    serial@f8b00000
Out:   serial@f8b00000
Err:   serial@f8b00000
Net:   Net Initialization Skipped
No ethernet found.

Hit any key to stop autoboot:  0
starting USB...
USB0:   USB EHCI 1.00
scanning bus 0 for devices... 1 USB Device(s) found
USB1:   USB EHCI 1.00
scanning bus 1 for devices... 4 USB Device(s) found
       scanning usb for storage devices... 1 Storage Device(s) found
       scanning usb for ethernet devices... 1 Ethernet Device(s) found

USB device 0:
    Device 0: Vendor: SanDisk Rev: 1.00 Prod: Cruzer Blade
            Type: Removable Hard Disk
            Capacity: 7632.0 MB = 7.4 GB (15630336 x 512)
... is now current device
Scanning usb 0:1...
=>
```

6.20 QEMU virt Armv8-A

Trusted Firmware-A (TF-A) implements the EL3 firmware layer for QEMU virt Armv8-A. BL1 is used as the BootROM, supplied with the -bios argument. When QEMU starts all CPUs are released simultaneously, BL1 selects a primary CPU to handle the boot and the secondaries are placed in a polling loop to be released by normal world via PSCI.

BL2 edits the Flattened Device Tree, FDT, generated by QEMU at run-time to add a node describing PSCI and also enable methods for the CPUs.

If `ARM_LINUX_KERNEL_AS_BL33` is set to 1 then this FDT will be passed to BL33 via register x0, as expected by a Linux kernel. This allows a Linux kernel image to be booted directly as BL33 rather than using a bootloader.

An ARM64 defconfig v5.5 Linux kernel is known to boot, FDT doesn't need to be provided as it's generated by QEMU.

Current limitations:

- Only cold boot is supported

6.20.1 Getting non-TF images

QEMU_EFI.fd can be downloaded from http://snapshots.linaro.org/components/kernel/leg-virt-tianocore-edk2-upstream/latest/QEMU-KERNEL-AARCH64/RELEASE_GCC5/QEMU_EFI.fd

or, can be built as follows:

```
git clone https://github.com/tianocore/edk2.git
cd edk2
git submodule update --init
make -C BaseTools
source edksetup.sh
export GCC5_AARCH64_PREFIX=aarch64-linux-gnu-
build -a AARCH64 -t GCC5 -p ArmVirtPkg/ArmVirtQemuKernel.dsc
```

Then, you will get `Build/ArmVirtQemuKernel-AARCH64/DEBUG_GCC5/FV/QEMU_EFI.fd`

Please note you do not need to use GCC 5 in spite of the environment variable `GCC5_AARCH64_PREFIX`

The rootfs can be built by using Buildroot as follows:

```
git clone git://git.buildroot.net/buildroot.git
cd buildroot
make qemu_aarch64_virt_defconfig
utils/config -e BR2_TARGET_ROOTFS_CPIO
utils/config -e BR2_TARGET_ROOTFS_CPIO_GZIP
make olddefconfig
make
```

Then, you will get `output/images/rootfs.cpio.gz`.

6.20.2 Booting via semi-hosting option

Boot binaries, except BL1, are primarily loaded via semi-hosting so all binaries has to reside in the same directory as QEMU is started from. This is conveniently achieved with symlinks the local names as:

- `bl2.bin` -> BL2
- `bl31.bin` -> BL31
- `bl33.bin` -> BL33 (`QEMU_EFI.fd`)
- `Image` -> linux/arch/arm64/boot/Image

To build:

```
make CROSS_COMPILE=aarch64-none-elf- PLAT=qemu
```

To start (QEMU v5.0.0):

```
qemu-system-aarch64 -nographic -machine virt,secure=on -cpu cortex-a57  \
    -kernel Image                           \
    -append "console=ttyAMA0,38400 keep_bootcon"   \
    -initrd rootfs.cpio.gz -smp 2 -m 1024 -bios bl1.bin   \
    -d unimp -semihosting-config enable,target=native
```

6.20.3 Booting via flash based firmwares

Boot firmwares are loaded via secure FLASH0 device so `bl1.bin` and `fip.bin` should be concatenated to create a `flash.bin` that is flashed onto secure FLASH0.

- `bl32.bin` -> BL32 (`tee-header_v2.bin`)
- `bl32_extra1.bin` -> BL32 Extra1 (`tee-pager_v2.bin`)
- `bl32_extra2.bin` -> BL32 Extra2 (`tee-pageable_v2.bin`)
- `bl33.bin` -> BL33 (`QEMU_EFI.fd`)
- `Image` -> linux/arch/arm64/boot/Image

To build:

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=qemu BL32=bl32.bin \
    BL32_EXTRA1=bl32_extra1.bin BL32_EXTRA2=bl32_extra2.bin \
    BL33=bl33.bin BL32_RAM_LOCATION=tdram SPD=opteed all fip
```

To build with TBBR enabled, BL31 and BL32 encrypted with test key:

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=qemu BL32=bl32.bin \
    BL32_EXTRA1=bl32_extra1.bin BL32_EXTRA2=bl32_extra2.bin \
    BL33=bl33.bin BL32_RAM_LOCATION=tdram SPD=opteed all fip \
    MBEDTLS_DIR=<path-to-mbedtls-repo> TRUSTED_BOARD_BOOT=1 \
    GENERATE_COT=1 DECRYPTION_SUPPORT=aes_gcm FW_ENC_STATUS=0 \
    ENCRYPT_BL31=1 ENCRYPT_BL32=1
```

To build flash.bin:

```
dd if=build/qemu/release/bl1.bin of=flash.bin bs=4096 conv=notrunc
dd if=build/qemu/release/fip.bin of=flash.bin seek=64 bs=4096 conv=notrunc
```

To start (QEMU v5.0.0):

```
qemu-system-aarch64 -nographic -machine virt,secure=on -cpu cortex-a57  \
    -kernel Image -no-acpi                          \
    -append 'console=ttyAMA0,38400 keep_bootcon'   \
    -initrd rootfs.cpio.gz -smp 2 -m 1024 -bios flash.bin   \
    -d unimp
```

6.21 QEMU SBSA Target

Trusted Firmware-A (TF-A) implements the EL3 firmware layer for QEMU SBSA Armv8-A. While running Qemu from command line, we need to supply two Flash images. First Secure BootRom is supplied by -pflash argument. This Flash image is made by EDK2 build system by composing BL1 and FIP. Second parameter for Qemu is responsible for Non-secure rom which also given with -pflash argument and contains of UEFI and EFI variables (also made by EDK2 build system). Semihosting is not used

When QEMU starts all CPUs are released simultaneously, BL1 selects a primary CPU to handle the boot and the secondaries are placed in a polling loop to be released by normal world via PSCI.

BL2 edits the FDT, generated by QEMU at run-time to add a node describing PSCI and also enable methods for the CPUs.

Current limitations:

- Only cold boot is supported

To build TF-A:

```
git clone https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git tfa
cd tfa
export CROSS_COMPILE=aarch64-none-elf-
make PLAT=qemu_sbsa all fip
```

To build TF-A with BL32 and SPM enabled(StandaloneMM as a Secure Payload):

```
git clone https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git tfa
cd tfa
export CROSS_COMPILE=aarch64-none-elf-
make PLAT=qemu_sbsa BL32=../STANDALONE_MM.fd SPM_MM=1 EL3_EXCEPTION_HANDLING=1 all fip
```

Images will be placed at build/qemu_sbsa/release (bl1.bin and fip.bin). Need to copy them into top directory for EDK2 compilation.

```
cp build/qemu_sbsa/release/bl1.bin ../
cp build/qemu_sbsa/release/fip.bin ../
```

Those images cannot be used by itself (no semihosing support). Flash images are built by EDK2 build system, refer to edk2-platform repo for full build instructions.

```
git clone https://github.com/tianocore/edk2-platforms.git
Platform/Qemu/SbsaQemu/Readme.md
```

6.22 Qualcomm Technologies, Inc.

Trusted Firmware-A (TF-A) implements the EL3 firmware layer for QTI SC7180.

6.22.1 Boot Trace

Bootrom –> BL1/BL2 –> BL31 –> BL33 –> Linux kernel

BL1/2 and BL33 can currently be supplied from Coreboot + Depthcharge

6.22.2 How to build

Code Locations

- Trusted Firmware-A: link

Build Procedure

QTI SoC expects TF-A's BL31 to get integrated with other boot software Coreboot, so only bl31.elf need to get build from the TF-A repository.

The build command looks like

> make CROSS_COMPILE=aarch64-linux-gnu- PLAT=sc7180 COREBOOT=1

update value of CROSS_COMPILE argument with your cross-compilation toolchain.

Additional QTISECLIB_PATH=<path to qtiseclib> can be added in build command. if QTISECLIB_PATH is not added in build command stub implementation of qtiseclib is picked. qtiseclib with stub implementation doesn't boot device. This was added to satisfy compilation.

QTISELIB for SC7180 is available at link

6.23 Raspberry Pi 3

The Raspberry Pi 3 is an inexpensive single-board computer that contains four Arm Cortex-A53 cores.

The following instructions explain how to use this port of the TF-A with the default distribution of Raspbian because that's the distribution officially supported by the Raspberry Pi Foundation. At the moment of writing this, the officially supported kernel is a AArch32 kernel. This doesn't mean that this port of TF-A can't boot a AArch64 kernel. The Linux tree fork maintained by the Foundation can be compiled for AArch64 by following the steps in *AArch64 kernel build instructions*.

IMPORTANT NOTE: This port isn't secure. All of the memory used is DRAM, which is available from both the Non-secure and Secure worlds. This port shouldn't be considered more than a prototype to play with and implement elements like PSCI to support the Linux kernel.

6.23.1 Design

The SoC used by the Raspberry Pi 3 is the Broadcom BCM2837. It is a SoC with a VideoCore IV that acts as primary processor (and loads everything from the SD card) and is located between all Arm cores and the DRAM. Check the Raspberry Pi 3 documentation for more information.

This explains why it is possible to change the execution state (AArch64/AArch32) depending on a few files on the SD card. We only care about the cases in which the cores boot in AArch64 mode.

The rules are simple:

- If a file called `kernel8.img` is located on the `boot` partition of the SD card, it will load it and execute in EL2 in AArch64. Basically, it executes a default AArch64 stub at address **0x0** that jumps to the kernel.
- If there is also a file called `armstub8.bin`, it will load it at address **0x0** (instead of the default stub) and execute it in EL3 in AArch64. All the cores are powered on at the same time and start at address **0x0**.

This means that we can use the default AArch32 kernel provided in the official Raspbian distribution by renaming it to `kernel8.img`, while TF-A and anything else we need is in `armstub8.bin`. This way we can forget about the default bootstrap code. When using a AArch64 kernel, it is only needed to make sure that the name on the SD card is `kernel8.img`.

Ideally, we want to load the kernel and have all cores available, which means that we need to make the secondary cores work in the way the kernel expects, as explained in *Secondary cores*. In practice, a small bootstrap is needed between TF-A and the kernel.

To get the most out of a AArch32 kernel, we want to boot it in Hypervisor mode in AArch32. This means that BL33 can't be in EL2 in AArch64 mode. The architecture specifies that AArch32 Hypervisor mode isn't present when AArch64 is used for EL2. When using a AArch64 kernel, it should simply start in EL2.

Placement of images

The file `armstub8.bin` contains BL1 and the FIP. It is needed to add padding between them so that the addresses they are loaded to match the ones specified when compiling TF-A. This is done automatically by the build system.

The device tree block is loaded by the VideoCore loader from an appropriate file, but we can specify the address it is loaded to in `config.txt`.

The file `kernel8.img` contains a kernel image that is loaded to the address specified in `config.txt`. The Linux kernel tree has information about how a AArch32 Linux kernel image is loaded in `Documentation/arm/Booting`:

```
The zImage may also be placed in system RAM and called there.  The
kernel should be placed in the first 128MiB of RAM.  It is recommended
that it is loaded above 32MiB in order to avoid the need to relocate
prior to decompression, which will make the boot process slightly
faster.
```

There are no similar restrictions for AArch64 kernels, as specified in the file `Documentation/arm64/booting.txt`.

This means that we need to avoid the first 128 MiB of RAM when placing the TF-A images (and specially the first 32 MiB, as they are directly used to place the uncompressed AArch32 kernel image. This way, both AArch32 and AArch64 kernels can be placed at the same address.

In the end, the images look like the following diagram when placed in memory. All addresses are Physical Addresses from the point of view of the Arm cores. Again, note that this is all just part of the same DRAM that goes from **0x00000000** to **0x3F000000**, it just has different names to simulate a real secure platform!

```
0x00000000 +-----------------+
           |       ROM       | BL1
0x00020000 +-----------------+
           |       FIP       |
0x00200000 +-----------------+
           |                 |
           |       ...       |
           |                 |
0x01000000 +-----------------+
           |       DTB       |  (Loaded by the VideoCore)
           +-----------------+
           |                 |
           |       ...       |
           |                 |
0x02000000 +-----------------+
           |     Kernel      |  (Loaded by the VideoCore)
           +-----------------+
           |                 |
           |       ...       |
           |                 |
0x10000000 +-----------------+
           |   Secure SRAM   | BL2, BL31
0x10100000 +-----------------+
           |   Secure DRAM   | BL32 (Secure payload)
```

(continues on next page)

(continued from previous page)

```
0x11000000 +-----------------+
           | Non-secure DRAM | BL33
           +-----------------+
           |                 |
           |       ...       |
           |                 |
0x3F000000 +-----------------+
           |       I/O       |
0x40000000 +-----------------+
```

The area between **0x10000000** and **0x11000000** has to be manually protected so that the kernel doesn't use it. The current port tries to modify the live DTB to add a memreserve region that reserves the previously mentioned area.

If this is not possible, the user may manually add `memmap=16M$256M` to the command line passed to the kernel in `cmdline.txt`. See the *Setup SD card* instructions to see how to do it. This system is strongly discouraged.

The last 16 MiB of DRAM can only be accessed by the VideoCore, that has different mappings than the Arm cores in which the I/O addresses don't overlap the DRAM. The memory reserved to be used by the VideoCore is always placed at the end of the DRAM, so this space isn't wasted.

Considering the 128 MiB allocated to the GPU and the 16 MiB allocated for TF-A, there are 880 MiB available for Linux.

Boot sequence

The boot sequence of TF-A is the usual one except when booting an AArch32 kernel. In that case, BL33 is booted in AArch32 Hypervisor mode so that it can jump to the kernel in the same mode and let it take over that privilege level. If BL33 was running in EL2 in AArch64 (as in the default bootflow of TF-A) it could only jump to the kernel in AArch32 in Supervisor mode.

The Linux kernel tree has instructions on how to jump to the Linux kernel in `Documentation/arm/Booting` and `Documentation/arm64/booting.txt`. The bootstrap should take care of this.

This port support a direct boot of the Linux kernel from the firmware (as a BL33 image). Alternatively, U-Boot or other bootloaders may be used.

Secondary cores

This port of the Trusted Firmware-A supports `PSCI_CPU_ON`, `PSCI_SYSTEM_RESET` and `PSCI_SYSTEM_OFF`. The last one doesn't really turn the system off, it simply reboots it and asks the VideoCore firmware to keep it in a low power mode permanently.

The kernel used by Raspbian doesn't have support for PSCI, so it is needed to use mailboxes to trap the secondary cores until they are ready to jump to the kernel. This mailbox is located at a different address in the AArch32 default kernel than in the AArch64 kernel.

Kernels with PSCI support can use the PSCI calls instead for a cleaner boot.

Also, this port of TF-A has another Trusted Mailbox in Shared BL RAM. During cold boot, all secondary cores wait in a loop until they are given given an address to jump to in this Mailbox (`bl31_warm_entrypoint`).

Once BL31 has finished and the primary core has jumped to the BL33 payload, it has to call `PSCI_CPU_ON` to release the secondary CPUs from the wait loop. The payload then makes them wait in another waitloop listening from messages from the kernel. When the primary CPU jumps into the kernel, it will send an address to the mailbox so that the secondary CPUs jump to it and are recognised by the kernel.

6.23.2 Build Instructions

To boot a AArch64 kernel, only the AArch64 toolchain is required.

To boot a AArch32 kernel, both AArch64 and AArch32 toolchains are required. The AArch32 toolchain is needed for the AArch32 bootstrap needed to load a 32-bit kernel.

The build system concatenates BL1 and the FIP so that the addresses match the ones in the memory map. The resulting file is `armstub8.bin`, located in the build folder (e.g. `build/rpi3/debug/armstub8.bin`). To know how to use this file, follow the instructions in *Setup SD card*.

The following build options are supported:

- `RPI3_BL33_IN_AARCH32`: This port can load a AArch64 or AArch32 BL33 image. By default this option is 0, which means that TF-A will jump to BL33 in EL2 in AArch64 mode. If set to 1, it will jump to BL33 in Hypervisor in AArch32 mode.
- `PRELOADED_BL33_BASE`: Used to specify the address of a BL33 binary that has been preloaded by any other system than using the firmware. `BL33` isn't needed in the build command line if this option is used. Specially useful because the file `kernel8.img` can be loaded anywhere by modifying the file `config.txt`. It doesn't have to contain a kernel, it could have any arbitrary payload.
- `RPI3_DIRECT_LINUX_BOOT`: Disabled by default. Set to 1 to enable the direct boot of the Linux kernel from the firmware. Option `RPI3_PRELOADED_DTB_BASE` is mandatory when the direct Linux kernel boot is used. Options `PRELOADED_BL33_BASE` will most likely be needed as well because it is unlikely that the kernel image will fit in the space reserved for BL33 images. This option can be combined with `RPI3_BL33_IN_AARCH32` in order to boot a 32-bit kernel. The only thing this option does is to set the arguments in registers x0-x3 or r0-r2 as expected by the kernel.
- `RPI3_PRELOADED_DTB_BASE`: Auxiliary build option needed when using `RPI3_DIRECT_LINUX_BOOT=1`. This option allows to specify the location of a DTB in memory.
- `RPI3_RUNTIME_UART`: Indicates whether the UART should be used at runtime or disabled. `-1` (default) disables the runtime UART. Any other value enables the default UART (currently UART1) for runtime messages.
- `RPI3_USE_UEFI_MAP`: Set to 1 to build ATF with the altername memory mapping required for an UEFI firmware payload. These changes are needed to be able to run Windows on ARM64. This option, which is disabled by default, results in the following memory mappings:

```
0x00000000 +-----------------+
           |       ROM       | BL1
0x00010000 +-----------------+
           |       DTB       | (Loaded by the VideoCore)
0x00020000 +-----------------+
           |       FIP       |
0x00030000 +-----------------+
           |                 |
           |  UEFI PAYLOAD   |
           |                 |
0x00200000 +-----------------+
           |   Secure SRAM   | BL2, BL31
0x00300000 +-----------------+
           |   Secure DRAM   | BL32 (Secure payload)
0x00400000 +-----------------+
           |                 |
           |                 |
           | Non-secure DRAM | BL33
           |                 |
           |                 |
```

(continues on next page)

(continued from previous page)

```
0x01000000 +-----------------+
           |                 |
           |       ...       |
           |                 |
0x3F000000 +-----------------+
           |       I/O       |
```

- BL32: This port can load and run OP-TEE. The OP-TEE image is optional. Please use the code from here. Build the Trusted Firmware with option `BL32=tee-header_v2.bin BL32_EXTRA1=tee-pager_v2.bin BL32_EXTRA2=tee-pageable_v2.bin` to put the binaries into the FIP.

> **Warning:** If OP-TEE is used it may be needed to add the following options to the Linux command line so that the USB driver doesn't use FIQs: `dwc_otg.fiq_enable=0 dwc_otg.fiq_fsm_enable=0 dwc_otg.nak_holdoff=0`. This will unfortunately reduce the performance of the USB driver. It is needed when using Raspbian, for example.

- `TRUSTED_BOARD_BOOT`: This port supports TBB. Set this option to 1 to enable it. In order to use TBB, you might want to set `GENERATE_COT=1` to let the contents of the FIP automatically signed by the build process. The ROT key will be generated and output to `rot_key.pem` in the build directory. It is able to set ROT_KEY to your own key in PEM format. Also in order to build, you need to clone mbed TLS from here. `MBEDTLS_DIR` must point at the mbed TLS source directory.
- `ENABLE_STACK_PROTECTOR`: Disabled by default. It uses the hardware RNG of the board.

The following is not currently supported:

- AArch32 for TF-A itself.
- `EL3_PAYLOAD_BASE`: The reason is that you can already load anything to any address by changing the file `armstub8.bin`, so there's no point in using TF-A in this case.
- `MULTI_CONSOLE_API=0`: The multi console API must be enabled. Note that the crash console uses the internal 16550 driver functions directly in order to be able to print error messages during early crashes before setting up the multi console API.

Building the firmware for kernels that don't support PSCI

This is the case for the 32-bit image of Raspbian, for example. 64-bit kernels always support PSCI, but they may not know that the system understands PSCI due to an incorrect DTB file.

First, clone and compile the 32-bit version of the Raspberry Pi 3 TF-A bootstrap. Choose the one needed for the architecture of your kernel.

Then compile TF-A. For a 32-bit kernel, use the following command line:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=rpi3          \
RPI3_BL33_IN_AARCH32=1                                   \
BL33=../rpi3-arm-tf-bootstrap/aarch32/el2-bootstrap.bin
```

For a 64-bit kernel, use this other command line:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=rpi3          \
BL33=../rpi3-arm-tf-bootstrap/aarch64/el2-bootstrap.bin
```

However, enabling PSCI support in a 64-bit kernel is really easy. In the repository Raspberry Pi 3 TF-A bootstrap there is a patch that can be applied to the Linux kernel tree maintained by the Raspberry Pi foundation. It modifes the DTS to tell the kernel to use PSCI. Once this patch is applied, follow the instructions in *AArch64 kernel build instructions* to get a working 64-bit kernel image and supporting files.

Building the firmware for kernels that support PSCI

For a 64-bit kernel:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=rpi3          \
PRELOADED_BL33_BASE=0x02000000                           \
RPI3_PRELOADED_DTB_BASE=0x01000000                       \
RPI3_DIRECT_LINUX_BOOT=1
```

For a 32-bit kernel:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=rpi3          \
PRELOADED_BL33_BASE=0x02000000                           \
RPI3_PRELOADED_DTB_BASE=0x01000000                       \
RPI3_DIRECT_LINUX_BOOT=1                                 \
RPI3_BL33_IN_AARCH32=1
```

6.23.3 AArch64 kernel build instructions

The following instructions show how to install and run a AArch64 kernel by using a SD card with the default Raspbian install as base. Skip them if you want to use the default 32-bit kernel.

Note that this system won't be fully 64-bit because all the tools in the filesystem are 32-bit binaries, but it's a quick way to get it working, and it allows the user to run 64-bit binaries in addition to 32-bit binaries.

1. Clone the Linux tree fork maintained by the Raspberry Pi Foundation. To speed things up, do a shallow clone of the desired branch.

```
git clone --depth=1 -b rpi-4.18.y https://github.com/raspberrypi/linux
cd linux
```

2. Configure and compile the kernel. Adapt the number after `-j` so that it is 1.5 times the number of CPUs in your computer. This may take some time to finish.

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcmrpi3_defconfig
make -j 6 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

3. Copy the kernel image and the device tree to the SD card. Replace the path by the corresponding path in your computers to the `boot` partition of the SD card.

```
cp arch/arm64/boot/Image /path/to/boot/kernel8.img
cp arch/arm64/boot/dts/broadcom/bcm2710-rpi-3-b.dtb /path/to/boot/
cp arch/arm64/boot/dts/broadcom/bcm2710-rpi-3-b-plus.dtb /path/to/boot/
```

4. Install the kernel modules. Replace the path by the corresponding path to the filesystem partition of the SD card on your computer.

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- \
INSTALL_MOD_PATH=/path/to/filesystem modules_install
```

5. Follow the instructions in *Setup SD card* except for the step of renaming the existing `kernel7.img` (we have already copied a AArch64 kernel).

6.23.4 Setup SD card

The instructions assume that you have an SD card with a fresh install of Raspbian (or that, at least, the `boot` partition is untouched, or nearly untouched). They have been tested with the image available in 2018-03-13.

1. Insert the SD card and open the `boot` partition.
2. Rename `kernel7.img` to `kernel8.img`. This tricks the VideoCore bootloader into booting the Arm cores in AArch64 mode, like TF-A needs, even though the kernel is not compiled for AArch64.
3. Copy `armstub8.bin` here. When `kernel8.img` is available, The VideoCore bootloader will look for a file called `armstub8.bin` and load it at address **0x0** instead of a predefined one.
4. To enable the serial port "Mini UART" in Linux, open `cmdline.txt` and add `console=serial0, 115200 console=tty1`.
5. Open `config.txt` and add the following lines at the end (`enable_uart=1` is only needed to enable debugging through the Mini UART):

```
enable_uart=1
kernel_address=0x02000000
device_tree_address=0x01000000
```

If you connect a serial cable to the Mini UART and your computer, and connect to it (for example, with `screen /dev/ttyUSB0 115200`) you should see some text. In the case of an AArch32 kernel, you should see something like this:

```
NOTICE:  Booting Trusted Firmware
NOTICE:  BL1: v1.4(release):v1.4-329-g61e94684-dirty
NOTICE:  BL1: Built : 00:09:25, Nov  6 2017
NOTICE:  BL1: Booting BL2
NOTICE:  BL2: v1.4(release):v1.4-329-g61e94684-dirty
NOTICE:  BL2: Built : 00:09:25, Nov  6 2017
NOTICE:  BL1: Booting BL31
NOTICE:  BL31: v1.4(release):v1.4-329-g61e94684-dirty
NOTICE:  BL31: Built : 00:09:25, Nov  6 2017
[    0.266484] bcm2835-aux-uart 3f215040.serial: could not get clk: -517

Raspbian GNU/Linux 9 raspberrypi ttyS0
raspberrypi login:
```

Just enter your credentials, everything should work as expected. Note that the HDMI output won't show any text during boot.

6.24 Raspberry Pi 4

The *Raspberry Pi 4* is an inexpensive single-board computer that contains four Arm Cortex-A72 cores. Also in contrast to previous Raspberry Pi versions this model has a GICv2 interrupt controller.

This port is a minimal port to support loading non-secure EL2 payloads such as a 64-bit Linux kernel. Other payloads such as U-Boot or EDK-II should work as well, but have not been tested at this point.

IMPORTANT NOTE: This port isn't secure. All of the memory used is DRAM, which is available from both the Non-secure and Secure worlds. The SoC does not seem to feature a secure memory controller of any kind, so portions of DRAM can't be protected properly from the Non-secure world.

6.24.1 Build Instructions

There are no real configuration options at this point, so there is only one universal binary (bl31.bin), which can be built with:

```
CROSS_COMPILE=aarch64-linux-gnu- make PLAT=rpi4 DEBUG=1
```

Copy the generated build/rpi4/debug/bl31.bin to the SD card, adding an entry starting with `armstub=`, then followed by the respective file name to `config.txt`. You should have AArch64 code in the file loaded as the "kernel", as BL31 will drop into AArch64/EL2 to the respective load address. arm64 Linux kernels are known to work this way.

Other options that should be set in `config.txt` to properly boot 64-bit kernels are:

```
enable_uart=1
arm_64bit=1
enable_gic=1
```

The BL31 code will patch the provided device tree blob in memory to advertise PSCI support, also will add a reserved-memory node to the DT to tell the non-secure payload to not touch the resident TF-A code.

If you connect a serial cable between the Mini UART and your computer, and connect to it (for example, with `screen /dev/ttyUSB0 115200`) you should see some text from BL31, followed by the output of the EL2 payload. The command line provided is read from the `cmdline.txt` file on the SD card.

6.24.2 TF-A port design

In contrast to the existing Raspberry Pi 3 port this one here is a BL31-only port, also it deviates quite a lot from the RPi3 port in many other ways. There is not so much difference between the two models, so eventually those two could be (more) unified in the future.

As with the previous models, the GPU and its firmware are the first entity to run after the SoC gets its power. The on-chip Boot ROM loads the next stage (bootcode.bin) from flash (EEPROM), which is again GPU code. This part knows how to access the MMC controller and how to parse a FAT filesystem, so it will load further compononents and configuration files from the first FAT partition on the SD card.

To accommodate this existing way of configuring and setting up the board, we use as much of this workflow as possible. If bootcode.bin finds a file called `armstub8.bin` on the SD card or it gets pointed to such code by finding a `armstub=` key in `config.txt`, it will load this file to the beginning of DRAM (address 0) and execute it in AArch64 EL3. But before doing that, it will also load a "kernel" and the device tree into memory. The load addresses have a default, but can also be changed by setting them in `config.txt`. If the GPU firmware finds a magic value in the armstub image file, it will put those two load addresses in memory locations near the beginning of memory, where TF-A code picks them up.

To keep things simple, we will just use the kernel load address as the BL33 entry point, also put the DTB address in the x0 register, as requested by the arm64 Linux kernel boot protocol. This does not necessarily mean that the EL2 payload needs to be a Linux kernel, a bootloader or any other kernel would work as well, as long as it can cope with having the DT address in register x0. If the payload has other means of finding the device tree, it could ignore this address as well.

6.25 Renesas R-Car

"R-Car" is the nickname for Renesas' system-on-chip (SoC) family for car information systems designed for the next-generation of automotive computing for the age of autonomous vehicles.

The scalable R-Car hardware platform and flexible software platform cover the full product range, from the premium class to the entry level. Plug-ins are available for multiple open-source software tools.

6.25.1 Renesas R-Car Gen3 evaluation boards:

	Standard	Low Cost Boards (LCB)
R-Car H3	• Salvator-X • Salvator-XS	• R-Car Starter Kit Premier
R-Car M3-W	• Salvator-X • Salvator-XS	• R-Car Starter Kit Pro
R-Car M3-N	• Salvator-X • Salvator-XS	
R-Car V3M	• Eagle	• Starter Kit
R-Car V3H	• Condor	• Starter Kit
R-Car D3	• Draak	

boards info

The current TF-A port has been tested on the R-Car H3 Salvator-X Soc_id r8a7795 revision ES1.1 (uses a Secure Payload Dispatcher)

```
ARM CA57 (ARMv8) 1.5 GHz quad core, with NEON/VFPv4, L1$ I/D
48K/32K, L2$ 2MB
ARM CA53 (ARMv8) 1.2 GHz quad core, with NEON/VFPv4, L1$ I/D 32K/32K,
L2$ 512K
Memory controller for LPDDR4-3200 4GB in 2 channels, each 64-bit wide
Two- and three-dimensional graphics engines,
Video processing units,
3 channels Display Output,
```

(continues on next page)

(continued from previous page)

```
6 channels Video Input,
SD card host interface,
USB3.0 and USB2.0 interfaces,
CAN interfaces
Ethernet AVB
PCI Express Interfaces
Memories
    INTERNAL 384KB SYSTEM RAM
    DDR 4 GB LPDDR4
    HYPERFLASH 64 MB HYPER FLASH (512 MBITS, 160 MHZ, 320 MBYTES/S)
    QSPI FLASH 16MB QSPI (128 MBITS,80 MHZ,80 MBYTES/S)1 HEADER QSPI
    MODULE
    EMMC 32 GB EMMC (HS400 240 MBYTES/S)
    MICROSD-CARD SLOT (SDR104 100 MBYTES/S)
```

6.25.2 Overview

On the rcar-gen3 the BOOTROM starts the cpu at EL3; for this port BL2 will therefore be entered at this exception level (the Renesas' ATF reference tree [1] resets into EL1 before entering BL2 - see its bl2.ld.S)

BL2 initializes DDR (and on some platforms i2c to interface to the PMIC) before determining the boot reason (cold or warm).

During suspend all CPUs are switched off and the DDR is put in backup mode (some kind of self-refresh mode). This means that BL2 is always entered in a cold boot scenario.

Once BL2 boots, it determines the boot reason, writes it to shared memory (BOOT_KIND_BASE) together with the BL31 parameters (PARAMS_BASE) and jumps to BL31.

To all effects, BL31 is as if it is being entered in reset mode since it still needs to initialize the rest of the cores; this is the reason behind using direct shared memory access to BOOT_KIND_BASE _and_ PARAMS_BASE instead of using registers to get to those locations (see el3_common_macros.S and bl31_entrypoint.S for the RESET_TO_BL31 use case).

Depending on the boot reason BL31 initializes the rest of the cores: in case of suspend, it uses a MBOX memory region to recover the program counters.

[1] https://github.com/renesas-rcar/arm-trusted-firmware

6.25.3 How to build

The TF-A build options depend on the target board so you will have to refer to those specific instructions. What follows is customized to the H3 SiP Salvator-X development system used in this port.

Build Tested:

RCAR_OPT="LSI=H3 RCAR_DRAM_SPLIT=1 RCAR_LOSSY_ENABLE=1" MBEDTLS_DIR=$mbedtls_src

$ MBEDTLS_DIR=$mbedtls_src_tree make clean bl2 bl31 rcar_layout_tool PLAT=rcar ${RCAR_OPT} SPD=opteed

System Tested:

- mbed_tls: git@github.com:ARMmbed/mbedtls.git [devel]

  commit 552754a6ee82bab25d1bdf28c8261a4518e65e4d Merge: 68dbc94 f34a4c1 Author: Simon Butcher <simon.butcher@arm.com> Date: Thu Aug 30 00:57:28 2018 +0100

- optee_os: https://github.com/BayLibre/optee_os

  Until it gets merged into OP-TEE, the port requires Renesas' Trusted Environment with a modification to support power management. commit 80105192cba9e704ebe8df7ab84095edc2922f84

  Author: Jorge Ramirez-Ortiz <jramirez@baylibre.com> Date: Thu Aug 30 16:49:49 2018 +0200 plat-rcar: cpu-suspend: handle the power level Signed-off-by: Jorge Ramirez-Ortiz <jramirez@baylibre.com>

- u-boot: The port has beent tested using mainline uboot.

  commit 4cdeda511f8037015b568396e6dcc3d8fb41e8c0 Author: Fabio Estevam <festevam@gmail.com> Date: Tue Sep 4 10:23:12 2018 -0300

- linux: The port has beent tested using mainline kernel.

  commit 7876320f88802b22d4e2daf7eb027dd14175a0f8 Author: Linus Torvalds <torvalds@linux-foundation.org> Date: Sun Sep 16 11:52:37 2018 -0700 Linux 4.19-rc4

TF-A Build Procedure

- Fetch all the above 4 repositories.
- Prepare the AARCH64 toolchain.
- Build u-boot using r8a7795_salvator-x_defconfig. Result: u-boot-elf.srec

```
make CROSS_COMPILE=aarch64-linux-gnu-
   r8a7795_salvator-x_defconfig

make CROSS_COMPILE=aarch64-linux-gnu-
```

- Build atf Result: bootparam_sa0.srec, cert_header_sa6.srec, bl2.srec, bl31.srec

```
RCAR_OPT="LSI=H3 RCAR_DRAM_SPLIT=1 RCAR_LOSSY_ENABLE=1"

MBEDTLS_DIR=$mbedtls_src_tree make clean bl2 bl31 rcar \
PLAT=rcar ${RCAR_OPT} SPD=opteed
```

- Build optee-os Result: tee.srec

```
make -j8 PLATFORM="rcar" CFG_ARM64_core=y
```

Install Procedure

- Boot the board in Mini-monitor mode and enable access to the Hyperflash.
- Use the XSL2 Mini-monitor utility to accept all the SREC ascii transfers over serial.

6.25.4 Boot trace

Notice that BL31 traces are not accessible via the console and that in order to verbose the BL2 output you will have to compile TF-A with LOG_LEVEL=50 and DEBUG=1

```
Initial Program Loader(CA57) Rev.1.0.22
NOTICE:  BL2: PRR is R-Car H3 Ver.1.1
NOTICE:  BL2: Board is Salvator-X Rev.1.0
NOTICE:  BL2: Boot device is HyperFlash(80MHz)
NOTICE:  BL2: LCM state is CM
NOTICE:  AVS setting succeeded. DVFS_SetVID=0x53
NOTICE:  BL2: DDR1600(rev.0.33)NOTICE:  [COLD_BOOT]NOTICE:  ..0
NOTICE:  BL2: DRAM Split is 4ch
NOTICE:  BL2: QoS is default setting(rev.0.37)
NOTICE:  BL2: Lossy Decomp areas
NOTICE:       Entry 0: DCMPAREACRAx:0x80000540 DCMPAREACRBx:0x570
NOTICE:       Entry 1: DCMPAREACRAx:0x40000000 DCMPAREACRBx:0x0
NOTICE:       Entry 2: DCMPAREACRAx:0x20000000 DCMPAREACRBx:0x0
NOTICE:  BL2: v2.0(release):v2.0-rc0-32-gbcda69a
NOTICE:  BL2: Built : 16:41:23, Oct  2 2018
NOTICE:  BL2: Normal boot
INFO:    BL2: Doing platform setup
INFO:    BL2: Loading image id 3
NOTICE:  BL2: dst=0xe6322000 src=0x8180000 len=512(0x200)
NOTICE:  BL2: dst=0x43f00000 src=0x8180400 len=6144(0x1800)
WARNING: r-car ignoring the BL31 size from certificate,using
RCAR_TRUSTED_SRAM_SIZE instead
INFO:    Loading image id=3 at address 0x44000000
NOTICE:  rcar_file_len: len: 0x0003e000
NOTICE:  BL2: dst=0x44000000 src=0x81c0000 len=253952(0x3e000)
INFO:    Image id=3 loaded: 0x44000000 - 0x4403e000
INFO:    BL2: Loading image id 4
INFO:    Loading image id=4 at address 0x44100000
NOTICE:  rcar_file_len: len: 0x00100000
NOTICE:  BL2: dst=0x44100000 src=0x8200000 len=1048576(0x100000)
INFO:    Image id=4 loaded: 0x44100000 - 0x44200000
INFO:    BL2: Loading image id 5
INFO:    Loading image id=5 at address 0x50000000
NOTICE:  rcar_file_len: len: 0x00100000
NOTICE:  BL2: dst=0x50000000 src=0x8640000 len=1048576(0x100000)
INFO:    Image id=5 loaded: 0x50000000 - 0x50100000
NOTICE:  BL2: Booting BL31
INFO:    Entry point address = 0x44000000
INFO:    SPSR = 0x3cd
VERBOSE: Argument #0 = 0xe6325578
VERBOSE: Argument #1 = 0x0
VERBOSE: Argument #2 = 0x0
VERBOSE: Argument #3 = 0x0
VERBOSE: Argument #4 = 0x0
VERBOSE: Argument #5 = 0x0
VERBOSE: Argument #6 = 0x0
```

(continues on next page)

(continued from previous page)

```
VERBOSE: Argument #7 = 0x0


U-Boot 2018.09-rc3-00028-g3711616 (Sep 27 2018 - 18:50:24 +0200)

CPU: Renesas Electronics R8A7795 rev 1.1
Model: Renesas Salvator-X board based on r8a7795 ES2.0+
DRAM:  3.5 GiB
Flash: 64 MiB
MMC:   sd@ee100000: 0, sd@ee140000: 1, sd@ee160000: 2
Loading Environment from MMC... OK
In:    serial@e6e88000
Out:   serial@e6e88000
Err:   serial@e6e88000
Net:   eth0: ethernet@e6800000
Hit any key to stop autoboot:  0
=>
```

6.26 Rockchip SoCs

Trusted Firmware-A supports a number of Rockchip ARM SoCs from both AARCH32 and AARCH64 fields.

This includes right now: - px30: Quad-Core Cortex-A53 - rk3288: Quad-Core Cortex-A17 (past A12) - rk3328: Quad-Core Cortex-A53 - rk3368: Octa-Core Cortex-A53 - rk3399: Hexa-Core Cortex-A53/A72

6.26.1 Boot Sequence

For AARCH32: Bootrom –> BL1/BL2 –> BL32 –> BL33 –> Linux kernel

For AARCH64: Bootrom –> BL1/BL2 –> BL31 –> BL33 –> Linux kernel

BL1/2 and BL33 can currently be supplied from either: - Coreboot + Depthcharge - U-Boot - either separately as TPL+SPL or only SPL

6.26.2 How to build

Rockchip SoCs expect TF-A's BL31 (AARCH64) or BL32 (AARCH32) to get integrated with other boot software like U-Boot or Coreboot, so only these images need to get build from the TF-A repository.

For AARCH64 architectures the build command looks like

> make CROSS_COMPILE=aarch64-linux-gnu- PLAT=rk3399 bl32

while AARCH32 needs a slightly different command

> make ARCH=aarch32 CROSS_COMPILE=arm-linux-gnueabihf- PLAT=rk3288 AARCH32_SP=sp_min bl32

Both need replacing the PLAT argument with the platform from above you want to build for and the CROSS_COMPILE argument with you cross- compilation toolchain.

6.26.3 How to deploy

Both upstream U-Boot and Coreboot projects contain instructions on where to put the built images during their respective build process. So after successfully building TF-A just follow their build instructions to continue.

6.27 Socionext UniPhier

Socionext UniPhier Armv8-A SoCs use Trusted Firmware-A (TF-A) as the secure world firmware, supporting BL2 and BL31.

UniPhier SoC family implements its internal boot ROM, which loads 64KB[1] image from a non-volatile storage to the on-chip SRAM, and jumps over to it. TF-A provides a special mode, BL2-AT-EL3, which enables BL2 to execute at EL3. It is useful for platforms with non-TF-A boot ROM, like UniPhier. Here, a problem is BL2 does not fit in the 64KB limit if *Trusted Board Boot (TBB)* is enabled. To solve this issue, Socionext provides a first stage loader called UniPhier BL. This loader runs in the on-chip SRAM, initializes the DRAM, expands BL2 there, and hands the control over to it. Therefore, all images of TF-A run in DRAM.

The UniPhier platform works with/without TBB. See below for the build process of each case. The image authentication for the UniPhier platform fully complies with the Trusted Board Boot Requirements (TBBR) specification.

The UniPhier BL does not implement the authentication functionality, that is, it can not verify the BL2 image by itself. Instead, the UniPhier BL assures the BL2 validity in a different way; BL2 is GZIP-compressed and appended to the UniPhier BL. The concatenation of the UniPhier BL and the compressed BL2 fits in the 64KB limit. The concatenated image is loaded by the internal boot ROM (and verified if the chip fuses are blown).

6.27.1 Boot Flow

1. The Boot ROM

   This is hard-wired ROM, so never corrupted. It loads the UniPhier BL (with compressed-BL2 appended) into the on-chip SRAM. If the SoC fuses are blown, the image is verified by the SoC's own method.

2. UniPhier BL

   This runs in the on-chip SRAM. After the minimum SoC initialization and DRAM setup, it decompresses the appended BL2 image into the DRAM, then jumps to the BL2 entry.

3. BL2 (at EL3)

   This runs in the DRAM. It extracts more images such as BL31, BL33 (optionally SCP_BL2, BL32 as well) from Firmware Image Package (FIP). If TBB is enabled, they are all authenticated by the standard mechanism of TF-A. After loading all the images, it jumps to the BL31 entry.

4. BL31, BL32, and BL33

   They all run in the DRAM. See *Firmware Design* for details.

[1] Some SoCs can load 80KB, but the software implementation must be aligned to the lowest common denominator.

6.27.2 Basic Build

BL2 must be compressed for the reason above. The UniPhier's platform makefile provides a build target `bl2_gzip` for this.

For a non-secure boot loader (aka BL33), U-Boot is well supported for UniPhier SoCs. The U-Boot image (`u-boot.bin`) must be built in advance. For the build procedure of U-Boot, refer to the document in the U-Boot project.

To build minimum functionality for UniPhier (without TBB):

```
make CROSS_COMPILE=<gcc-prefix> PLAT=uniphier BL33=<path-to-BL33> bl2_gzip fip
```

Output images:

- `bl2.bin.gz`
- `fip.bin`

6.27.3 Optional features

- Trusted Board Boot

  mbed TLS is needed as the cryptographic and image parser modules. Refer to the *Prerequisites* document for the appropriate version of mbed TLS.

  To enable TBB, add the following options to the build command:

  ```
  TRUSTED_BOARD_BOOT=1 GENERATE_COT=1 MBEDTLS_DIR=<path-to-mbedtls>
  ```

- System Control Processor (SCP)

  If desired, FIP can include an SCP BL2 image. If BL2 finds an SCP BL2 image in FIP, BL2 loads it into DRAM and kicks the SCP. Most of UniPhier boards still work without SCP, but SCP provides better power management support.

  To include SCP BL2, add the following option to the build command:

  ```
  SCP_BL2=<path-to-SCP>
  ```

- BL32 (Secure Payload)

  To enable BL32, add the following options to the build command:

  ```
  SPD=<spd> BL32=<path-to-BL32>
  ```

  If you use TSP for BL32, `BL32=<path-to-BL32>` is not required. Just add the following:

  ```
  SPD=tspd
  ```

6.28 Socionext Synquacer

Socionext's Synquacer SC2A11 is a multi-core processor with 24 cores of Arm Cortex-A53. The Developerbox, of 96boards, is a platform that contains this processor. This port of the Trusted Firmware only supports this platform at the moment.

More information are listed in link.

6.28.1 How to build

Code Locations

- Trusted Firmware-A: link
- edk2: link
- edk2-platforms: link
- edk2-non-osi: link

Boot Flow

SCP firmware –> TF-A BL31 –> UEFI(edk2)

Build Procedure

- Firstly, in addition to the "normal" build tools you will also need a few specialist tools. On a Debian or Ubuntu operating system try:

```
sudo apt install acpica-tools device-tree-compiler uuid-dev
```

- Secondly, create a new working directory and store the absolute path to this directory in an environment variable, WORKSPACE. It does not matter where this directory is created but as an example:

```
export WORKSPACE=$HOME/build/developerbox-firmware
mkdir -p $WORKSPACE
```

- Run the following commands to clone the source code:

```
cd $WORKSPACE
git clone https://github.com/ARM-software/arm-trusted-firmware -b master
git clone https://github.com/tianocore/edk2.git -b master
git clone https://github.com/tianocore/edk2-platforms.git -b master
git clone https://github.com/tianocore/edk2-non-osi.git -b master
```

- Build ATF:

```
cd $WORKSPACE/arm-trusted-firmware
make -j`nproc` PLAT=synquacer PRELOADED_BL33_BASE=0x8200000 bl31 fiptool
tools/fiptool/fiptool create \
      --tb-fw ./build/synquacer/release/bl31.bin \
      --soc-fw ./build/synquacer/release/bl31.bin \
      --scp-fw ./build/synquacer/release/bl31.bin \
      ../edk2-non-osi/Platform/Socionext/DeveloperBox/fip_all_arm_tf.bin
```

- Build EDK2:

```
cd $WORKSPACE
export PACKAGES_PATH=$WORKSPACE/edk2:$WORKSPACE/edk2-platforms:$WORKSPACE/edk2-
↪non-osi
export ACTIVE_PLATFORM="Platform/Socionext/DeveloperBox/DeveloperBox.dsc"
export GCC5_AARCH64_PREFIX=aarch64-linux-gnu-
unset ARCH

. edk2/edksetup.sh
make -C edk2/BaseTools

build -p $ACTIVE_PLATFORM -b RELEASE -a AARCH64 -t GCC5 -n `nproc` -D DO_
↪X86EMU=TRUE
```

- The firmware image, which comprises the option ROM, ARM trusted firmware and EDK2 itself, can be found $WORKSPACE/../Build/DeveloperBox/RELEASE_GCC5/FV/. Use SYNQUACERFIRMWAREUPDATECAPSULEFMPPKCS7.Cap for UEFI capsule update and SPI_NOR_IMAGE.fd for the serial flasher.

  Note #1: -t GCC5 can be loosely translated as "enable link-time-optimization"; any version of gcc >= 5 will support this feature and may be used to build EDK2.

  Note #2: Replace -b RELEASE with -b DEBUG to build a debug.

Install the System Firmware

- Providing your Developerbox is fully working and has on operating system installed then you can adopt your the newly compiled system firmware using the capsule update method:.

```
sudo apt install fwupdate
sudo fwupdate --apply {50b94ce5-8b63-4849-8af4-ea479356f0e3} \
              SYNQUACERFIRMWAREUPDATECAPSULEFMPPKCS7.Cap
sudo reboot
```

- Alternatively you can install SPI_NOR_IMAGE.fd using the board recovery method.

6.29 STMicroelectronics STM32MP1

STM32MP1 is a microprocessor designed by STMicroelectronics based on a dual Arm Cortex-A7. It is an Armv7-A platform, using dedicated code from TF-A. The STM32MP1 chip also embeds a Cortex-M4. More information can be found on STM32MP1 Series page.

6.29.1 STM32MP1 Versions

The STM32MP1 series is available in 3 different lines which are pin-to-pin compatible:

- STM32MP157: Dual Cortex-A7 cores, Cortex-M4 core @ 209 MHz, 3D GPU, DSI display interface and CAN FD
- STM32MP153: Dual Cortex-A7 cores, Cortex-M4 core @ 209 MHz and CAN FD
- STM32MP151: Single Cortex-A7 core, Cortex-M4 core @ 209 MHz

Each line comes with a security option (cryptography & secure boot) and a Cortex-A frequency option:

- A Basic + Cortex-A7 @ 650 MHz

- C Secure Boot + HW Crypto + Cortex-A7 @ 650 MHz
- D Basic + Cortex-A7 @ 800 MHz
- F Secure Boot + HW Crypto + Cortex-A7 @ 800 MHz

The STM32MP1 part number codification page gives more information about part numbers.

6.29.2 Design

The STM32MP1 resets in the ROM code of the Cortex-A7. The primary boot core (core 0) executes the boot sequence while secondary boot core (core 1) is kept in a holding pen loop. The ROM code boot sequence loads the TF-A binary image from boot device to embedded SRAM.

The TF-A image must be properly formatted with a STM32 header structure for ROM code is able to load this image. Tool stm32image can be used to prepend this header to the generated TF-A binary.

At compilation step, BL2, BL32 and DTB file are linked together in a single binary. The stm32image tool is also generated and the header is added to TF-A binary. This binary file with header is named tf-a-stm32mp157c-ev1.stm32. It can then be copied in the first partition of the boot device.

Memory mapping

```
0x00000000 +-----------------+
           |                 |   ROM
0x00020000 +-----------------+
           |                 |
           |       ...       |
           |                 |
0x2FFC0000 +-----------------+ \
           |                 | |
           |       ...       | |
           |                 | |
0x2FFD8000 +-----------------+ |
           |    TF-A DTB     | | Embedded SRAM
0x2FFDC000 +-----------------+ |
           |       BL2       | |
0x2FFEF000 +-----------------+ |
           |       BL32      | |
0x30000000 +-----------------+ /
           |                 |
           |       ...       |
           |                 |
0x40000000 +-----------------+
           |                 |
           |                 |   Devices
           |                 |
0xC0000000 +-----------------+ \
           |                 | |
0xC0100000 +-----------------+ |
           |       BL33      | | Non-secure RAM (DDR)
           |       ...       | |
           |                 | |
0xFFFFFFFF +-----------------+ /
```

Boot sequence

ROM code -> BL2 (compiled with BL2_AT_EL3) -> BL32 (SP_min) -> BL33 (U-Boot)

or if Op-TEE is used:

ROM code -> BL2 (compiled with BL2_AT_EL3) -> OP-TEE -> BL33 (U-Boot)

6.29.3 Build Instructions

Boot media(s) supported by BL2 must be specified in the build command. Available storage medias are: - `STM32MP_SDMMC` - `STM32MP_EMMC` - `STM32MP_RAW_NAND` - `STM32MP_SPI_NAND` - `STM32MP_SPI_NOR`

To build with SP_min and support for all bootable devices:

```
make CROSS_COMPILE=arm-linux-gnueabihf- PLAT=stm32mp1 ARCH=aarch32 ARM_ARCH_MAJOR=7␣
↪AARCH32_SP=sp_min STM32MP_SDMMC=1 STM32MP_EMMC=1 STM32MP_RAW_NAND=1 STM32MP_SPI_
↪NAND=1
STM32MP_SPI_NOR=1 DTB_FILE_NAME=stm32mp157c-ev1.dtb
cd <u-boot_directory>
make stm32mp15_trusted_defconfig
make DEVICE_TREE=stm32mp157c-ev1 all
```

To build TF-A with OP-TEE support for all bootable devices: .. code:: bash

> make CROSS_COMPILE=arm-linux-gnueabihf- PLAT=stm32mp1 ARCH=aarch32 ARM_ARCH_MAJOR=7 AARCH32_SP=optee STM32MP_SDMMC=1 STM32MP_EMMC=1 STM32MP_RAW_NAND=1 STM32MP_SPI_NAND=1 STM32MP_SPI_NOR=1 DTB_FILE_NAME=stm32mp157c-ev1.dtb cd <optee_directory> make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm PLATFORM=stm32mp1 CFG_EMBED_DTB_SOURCE_FILE=stm32mp157c-ev1.dts cd <u-boot_directory> make stm32mp15_trusted_defconfig make DEVICE_TREE=stm32mp157c-ev1 all

The following build options are supported:

- `ENABLE_STACK_PROTECTOR`: To enable the stack protection.

6.29.4 Populate SD-card

The SD-card has to be formated with GPT. It should contain at least those partitions:

- fsbl: to copy the tf-a-stm32mp157c-ev1.stm32 binary
- ssbl: to copy the u-boot.stm32 binary

Usually, two copies of fsbl are used (fsbl1 and fsbl2) instead of one partition fsbl.

OP-TEE artifacts go into separate partitions as follows:

- teeh: tee-header_v2.stm32
- teed: tee-pageable_v2.stm32
- teex: tee-pager_v2.stm32

6.30 Texas Instruments K3

Trusted Firmware-A (TF-A) implements the EL3 firmware layer for Texas Instruments K3 SoCs.

6.30.1 Boot Flow

```
R5(U-Boot) --> TF-A BL31 --> BL32(OP-TEE) --> TF-A BL31 --> BL33(U-Boot) --> Linux
                                                      \
                                                Optional direct to Linux boot
                                                          \
                                                          --> BL33(Linux)
```

Texas Instruments K3 SoCs contain an R5 processor used as the boot master, it loads the needed images for A53 startup, because of this we do not need BL1 or BL2 TF-A stages.

6.30.2 Build Instructions

https://github.com/ARM-software/arm-trusted-firmware.git

TF-A:

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=k3 SPD=opteed all
```

OP-TEE:

```
make ARCH=arm CROSS_COMPILE64=aarch64-linux-gnu- PLATFORM=k3 CFG_ARM64_core=y all
```

R5 U-Boot:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- am65x_evm_r5_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- SYSFW=<path to SYSFW>
```

A53 U-Boot:

```
make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- am65x_evm_a53_defconfig
make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- ATF=<path> TEE=<path>
```

6.30.3 Deploy Images

```
cp tiboot3.bin tispl.bin u-boot.img /sdcard/boot/
```

6.31 Xilinx Versal

Trusted Firmware-A implements the EL3 firmware layer for Xilinx Versal. The platform only uses the runtime part of TF-A as Xilinx Versal already has a BootROM (BL1) and PMC FW (BL2).

BL31 is TF-A. BL32 is an optional Secure Payload. BL33 is the non-secure world software (U-Boot, Linux etc).

To build: `` `bash make RESET_TO_BL31=1 CROSS_COMPILE=aarch64-none-elf- PLAT=versal bl31 ` ``

To build ATF for different platform (supported are "silicon"(default) and "versal_virt") `` `bash make RESET_TO_BL31=1 CROSS_COMPILE=aarch64-none-elf- PLAT=versal VERSAL_PLATFORM=versal_virt bl31 ` ``

6.31.1 Xilinx Versal platform specific build options

- *VERSAL_ATF_MEM_BASE*: Specifies the base address of the bl31 binary.
- *VERSAL_ATF_MEM_SIZE*: Specifies the size of the memory region of the bl31 binary.
- *VERSAL_BL32_MEM_BASE*: Specifies the base address of the bl32 binary.
- *VERSAL_BL32_MEM_SIZE*: Specifies the size of the memory region of the bl32 binary.
- *VERSAL_CONSOLE*: Select the console driver. Options: - *pl011*, *pl011_0*: ARM pl011 UART 0 - *pl011_1* : ARM pl011 UART 1
- *VERSAL_PLATFORM*: Select the platform. Options: - *versal_virt* : Versal Virtual platform

6.31.2 # PLM->TF-A Parameter Passing

The PLM populates a data structure with image information for the TF-A. The TF-A uses that data to hand off to the loaded images. The address of the handoff data structure is passed in the `` `PMC_GLOBAL_GLOB_GEN_STORAGE4` `` register. The register is free to be used by other software once the TF-A is bringing up further firmware images.

6.32 Xilinx Zynq UltraScale+ MPSoC

Trusted Firmware-A (TF-A) implements the EL3 firmware layer for Xilinx Zynq UltraScale + MPSoC. The platform only uses the runtime part of TF-A as ZynqMP already has a BootROM (BL1) and FSBL (BL2).

BL31 is TF-A. BL32 is an optional Secure Payload. BL33 is the non-secure world software (U-Boot, Linux etc).

To build:

```
make CROSS_COMPILE=aarch64-none-elf- PLAT=zynqmp bl31
```

To build bl32 TSP you have to rebuild bl31 too:

```
make CROSS_COMPILE=aarch64-none-elf- PLAT=zynqmp SPD=tspd bl31 bl32
```

6.32.1 ZynqMP platform specific build options

- `ZYNQMP_ATF_MEM_BASE`: Specifies the base address of the bl31 binary.
- `ZYNQMP_ATF_MEM_SIZE`: Specifies the size of the memory region of the bl31 binary.
- `ZYNQMP_BL32_MEM_BASE`: Specifies the base address of the bl32 binary.
- `ZYNQMP_BL32_MEM_SIZE`: Specifies the size of the memory region of the bl32 binary.
- `ZYNQMP_CONSOLE`: Select the console driver. Options:
  - `cadence`, `cadence0`: Cadence UART 0
  - `cadence1` : Cadence UART 1

6.32.2 FSBL->TF-A Parameter Passing

The FSBL populates a data structure with image information for TF-A. TF-A uses that data to hand off to the loaded images. The address of the handoff data structure is passed in the `PMU_GLOBAL.GLOBAL_GEN_STORAGE6` register. The register is free to be used by other software once TF-A has brought up further firmware images.

6.32.3 Power Domain Tree

The following power domain tree represents the power domain model used by TF-A for ZynqMP:

```
              +-+
              |0|
              +-+
 +-------+---+---+-------+
 |       |       |       |
 |       |       |       |
 v       v       v       v
+-+     +-+     +-+     +-+
|0|     |1|     |2|     |3|
+-+     +-+     +-+     +-+
```

The 4 leaf power domains represent the individual A53 cores, while resources common to the cluster are grouped in the power domain on the top.

6.33 Broadcom Stingray

6.33.1 Description

Broadcom's Stingray(BCM958742t) is a multi-core processor with 8 Cortex-A72 cores. Trusted Firmware-A (TF-A) is used to implement secure world firmware, supporting BL2 and BL31 for Broadcom Stingray SoCs.

On Poweron, Boot ROM will load bl2 image and Bl2 will initialize the hardware, then loads bl31 and bl33 into DDR and boots to bl33.

6.33.2 Boot Sequence

Bootrom –> TF-A BL2 –> TF-A BL31 –> BL33(u-boot)

Code Locations

- Trusted Firmware-A: link

6.33.3 How to build

Build Procedure

- Prepare AARCH64 toolchain.
- Build u-boot first, and get the binary image: u-boot.bin,
- Build TF-A

  Build fip:

```
make CROSS_COMPILE=aarch64-linux-gnu- PLAT=stingray BOARD_CFG=bcm958742t all fip␣
↪BL33=u-boot.bin
```

Deploy TF-A Images

The u-boot will be upstreamed soon, this doc will be updated once they are ready, and the link will be posted.

This section provides a list of supported upstream *platform ports* and the documentation associated with them.

Note: In addition to the platforms ports listed within the table of contents, there are several additional platforms that are supported upstream but which do not currently have associated documentation:

- Arm Neoverse N1 System Development Platform (N1SDP)
- Arm Neoverse Reference Design N1 Edge (RD-N1-Edge) FVP
- Arm Neoverse Reference Design E1 Edge (RD-E1-Edge) FVP
- Arm SGI-575 and SGM-775
- MediaTek MT6795 and MT8173 SoCs
- Arm Morello Platform

CHAPTER SEVEN

PERFORMANCE & TESTING

7.1 PSCI Performance Measurements on Arm Juno Development Platform

This document summarises the findings of performance measurements of key operations in the Trusted Firmware-A Power State Coordination Interface (PSCI) implementation, using the in-built Performance Measurement Framework (PMF) and runtime instrumentation timestamps.

7.1.1 Method

We used the Juno R1 platform for these tests, which has 4 x Cortex-A53 and 2 x Cortex-A57 clusters running at the following frequencies:

Domain	Frequency (MHz)
Cortex-A57	900 (nominal)
Cortex-A53	650 (underdrive)
AXI subsystem	533

Juno supports CPU, cluster and system power down states, corresponding to power levels 0, 1 and 2 respectively. It does not support any retention states.

We used the upstream TF master as of 31/01/2017, building the platform using the `ENABLE_RUNTIME_INSTRUMENTATION` option:

```
make PLAT=juno ENABLE_RUNTIME_INSTRUMENTATION=1 \
    SCP_BL2=<path/to/scp-fw.bin>                \
    BL33=<path/to/test-fw.bin>                  \
    all fip
```

When using the debug build of TF, there was no noticeable difference in the results.

The tests are based on an ARM-internal test framework. The release build of this framework was used because the results in the debug build became skewed; the console output prevented some of the tests from executing in parallel.

The tests consist of both parallel and sequential tests, which are broadly described as follows:

- **Parallel Tests** This type of test powers on all the non-lead CPUs and brings them and the lead CPU to a common synchronization point. The lead CPU then initiates the test on all CPUs in parallel.
- **Sequential Tests** This type of test powers on each non-lead CPU in sequence. The lead CPU initiates the test on a non-lead CPU then waits for the test to complete before proceeding to the next non-lead CPU. The lead CPU then executes the test on itself.

In the results below, CPUs 0-3 refer to CPUs in the little cluster (A53) and CPUs 4-5 refer to CPUs in the big cluster (A57). In all cases CPU 4 is the lead CPU.

`PSCI_ENTRY` refers to the time taken from entering the TF PSCI implementation to the point the hardware enters the low power state (WFI). Referring to the TF runtime instrumentation points, this corresponds to: `(RT_INSTR_ENTER_HW_LOW_PWR - RT_INSTR_ENTER_PSCI)`.

`PSCI_EXIT` refers to the time taken from the point the hardware exits the low power state to exiting the TF PSCI implementation. This corresponds to: `(RT_INSTR_EXIT_PSCI - RT_INSTR_EXIT_HW_LOW_PWR)`.

`CFLUSH_OVERHEAD` refers to the part of `PSCI_ENTRY` taken to flush the caches. This corresponds to: `(RT_INSTR_EXIT_CFLUSH - RT_INSTR_ENTER_CFLUSH)`.

Note there is very little variance observed in the values given (~1us), although the values for each CPU are sometimes interchanged, depending on the order in which locks are acquired. Also, there is very little variance observed between executing the tests sequentially in a single boot or rebooting between tests.

Given that runtime instrumentation using PMF is invasive, there is a small (unquantified) overhead on the results. PMF uses the generic counter for timestamps, which runs at 50MHz on Juno.

7.1.2 Results and Commentary

`CPU_SUSPEND` to deepest power level on all CPUs in parallel

CPU	`PSCI_ENTRY` (us)	`PSCI_EXIT` (us)	`CFLUSH_OVERHEAD` (us)
0	27	20	5
1	114	86	5
2	202	58	5
3	375	29	94
4	20	22	6
5	290	18	206

A large variance in `PSCI_ENTRY` and `PSCI_EXIT` times across CPUs is observed due to TF PSCI lock contention. In the worst case, CPU 3 has to wait for the 3 other CPUs in the cluster (0-2) to complete `PSCI_ENTRY` and release the lock before proceeding.

The `CFLUSH_OVERHEAD` times for CPUs 3 and 5 are higher because they are the last CPUs in their respective clusters to power down, therefore both the L1 and L2 caches are flushed.

The `CFLUSH_OVERHEAD` time for CPU 5 is a lot larger than that for CPU 3 because the L2 cache size for the big cluster is lot larger (2MB) compared to the little cluster (1MB).

`CPU_SUSPEND` to power level 0 on all CPUs in parallel

CPU	`PSCI_ENTRY` (us)	`PSCI_EXIT` (us)	`CFLUSH_OVERHEAD` (us)
0	116	14	8
1	204	14	8
2	287	13	8
3	376	13	9
4	29	15	7
5	21	15	8

There is no lock contention in TF generic code at power level 0 but the large variance in `PSCI_ENTRY` times across CPUs is due to lock contention in Juno platform code. The platform lock is used to mediate access to a single SCP communication channel. This is compounded by the SCP firmware waiting for each AP CPU to enter WFI before making the channel available to other CPUs, which effectively serializes the SCP power down commands from all CPUs.

On platforms with a more efficient CPU power down mechanism, it should be possible to make the `PSCI_ENTRY` times smaller and consistent.

The `PSCI_EXIT` times are consistent across all CPUs because TF does not require locks at power level 0.

The `CFLUSH_OVERHEAD` times for all CPUs are small and consistent since only the cache associated with power level 0 is flushed (L1).

`CPU_SUSPEND` to deepest power level on all CPUs in sequence

CPU	`PSCI_ENTRY` (us)	`PSCI_EXIT` (us)	`CFLUSH_OVERHEAD` (us)
0	114	20	94
1	114	20	94
2	114	20	94
3	114	20	94
4	195	22	180
5	21	17	6

The `CFLUSH_OVERHEAD` times for lead CPU 4 and all CPUs in the non-lead cluster are large because all other CPUs in the cluster are powered down during the test. The `CPU_SUSPEND` call powers down to the cluster level, requiring a flush of both L1 and L2 caches.

The `CFLUSH_OVERHEAD` time for CPU 4 is a lot larger than those for the little CPUs because the L2 cache size for the big cluster is lot larger (2MB) compared to the little cluster (1MB).

The `PSCI_ENTRY` and `CFLUSH_OVERHEAD` times for CPU 5 are low because lead CPU 4 continues to run while CPU 5 is suspended. Hence CPU 5 only powers down to level 0, which only requires L1 cache flush.

`CPU_SUSPEND` to power level 0 on all CPUs in sequence

CPU	`PSCI_ENTRY` (us)	`PSCI_EXIT` (us)	`CFLUSH_OVERHEAD` (us)
0	22	14	5
1	22	14	5
2	21	14	5
3	22	14	5
4	17	14	6
5	18	15	6

Here the times are small and consistent since there is no contention and it is only necessary to flush the cache to power level 0 (L1). This is the best case scenario.

The `PSCI_ENTRY` times for CPUs in the big cluster are slightly smaller than for the CPUs in little cluster due to greater CPU performance.

The `PSCI_EXIT` times are generally lower than in the last test because the cluster remains powered on throughout the test and there is less code to execute on power on (for example, no need to enter CCI coherency)

`CPU_OFF` on all non-lead CPUs in sequence then `CPU_SUSPEND` on lead CPU to deepest power level

The test sequence here is as follows:

1. Call `CPU_ON` and `CPU_OFF` on each non-lead CPU in sequence.
2. Program wake up timer and suspend the lead CPU to the deepest power level.
3. Call `CPU_ON` on non-lead CPU to get the timestamps from each CPU.

CPU	`PSCI_ENTRY` (us)	`PSCI_EXIT` (us)	`CFLUSH_OVERHEAD` (us)
0	110	28	93
1	110	28	93
2	110	28	93
3	111	28	93
4	195	22	181
5	20	23	6

The `CFLUSH_OVERHEAD` times for all little CPUs are large because all other CPUs in that cluster are powerered down during the test. The `CPU_OFF` call powers down to the cluster level, requiring a flush of both L1 and L2 caches.

The `PSCI_ENTRY` and `CFLUSH_OVERHEAD` times for CPU 5 are small because lead CPU 4 is running and CPU 5 only powers down to level 0, which only requires an L1 cache flush.

The `CFLUSH_OVERHEAD` time for CPU 4 is a lot larger than those for the little CPUs because the L2 cache size for the big cluster is lot larger (2MB) compared to the little cluster (1MB).

The `PSCI_EXIT` times for CPUs in the big cluster are slightly smaller than for CPUs in the little cluster due to greater CPU performance. These times generally are greater than the `PSCI_EXIT` times in the `CPU_SUSPEND` tests because there is more code to execute in the "on finisher" compared to the "suspend finisher" (for example, GIC redistributor register programming).

`PSCI_VERSION` on all CPUs in parallel

Since very little code is associated with `PSCI_VERSION`, this test approximates the round trip latency for handling a fast SMC at EL3 in TF.

CPU	TOTAL TIME (ns)
0	3020
1	2940
2	2980
3	3060
4	520
5	720

The times for the big CPUs are less than the little CPUs due to greater CPU performance.

We suspect the time for lead CPU 4 is shorter than CPU 5 due to subtle cache effects, given that these measurements are at the nano-second level.

7.2 Test Secure Payload (TSP) and Dispatcher (TSPD)

7.2.1 Building the Test Secure Payload

The TSP is coupled with a companion runtime service in the BL31 firmware, called the TSPD. Therefore, if you intend to use the TSP, the BL31 image must be recompiled as well. For more information on SPs and SPDs, see the *Secure-EL1 Payloads and Dispatchers* section in the *Firmware Design*.

First clean the TF-A build directory to get rid of any previous BL31 binary. Then to build the TSP image use:

```
make PLAT=<platform> SPD=tspd all
```

An additional boot loader binary file is created in the `build` directory:

```
build/<platform>/<build-type>/bl32.bin
```

7.3 Performance Monitoring Unit

The Performance Monitoring Unit (PMU) allows recording of architectural and microarchitectural events for profiling purposes.

This document gives an overview of the PMU counter configuration to assist with implementation and to complement the PMU security guidelines given in the *Secure Development Guidelines* document.

Note: This section applies to Armv8-A implementations which have version 3 of the Performance Monitors Extension (PMUv3).

7.3.1 PMU Counters

The PMU makes 32 counters available at all privilege levels:

- 31 programmable event counters: `PMEVCNTR<n>`, where `n` is `0` to `30`.
- A dedicated cycle counter: `PMCCNTR`.

Architectural mappings

Counters	State	System Register Name
Programmable	AArch64	`PMEVCNTR<n>_EL0[63*:0]`
	AArch32	`PMEVCNTR<n>[31:0]`
Cycle	AArch64	`PMCCNTR_EL0[63:0]`
	AArch32	`PMCCNTR[63:0]`

Note: Bits [63:32] are only available if ARMv8.5-PMU is implemented. Refer to the Arm ARM for a detailed description of ARMv8.5-PMU features.

7.3.2 Configuring the PMU for counting events

Each programmable counter has an associated register, `PMEVTYPER<n>` which configures it. The cycle counter has the `PMCCFILTR_EL0` register, which has an identical function and bit field layout as `PMEVTYPER<n>`. In addition, the counters are enabled (permitted to increment) via the `PMCNTENSET` and `PMCR` registers. These can be accessed at all privilege levels.

Architectural mappings

AArch64	AArch32
`PMEVTYPER<n>_EL0[63*:0]`	`PMEVTYPER<n>[31:0]`
`PMCCFILTR_EL0[63*:0]`	`PMCCFILTR[31:0]`
`PMCNTENSET_EL0[63*:0]`	`PMCNTENSET[31:0]`
`PMCR_EL0[63*:0]`	`PMCR[31:0]`

Note: Bits [63:32] are reserved.

Relevant register fields

For `PMEVTYPER<n>_EL0`/`PMEVTYPER<n>` and `PMCCFILTR_EL0/PMCCFILTR`, the most important fields are:

- `P`:
  - Bit 31.
  - If set to `0`, will increment the associated `PMEVCNTR<n>` at EL1.
- `NSK`:
  - Bit 29.
  - If equal to the `P` bit it enables the associated `PMEVCNTR<n>` at Non-secure EL1.
  - Reserved if EL3 not implemented.
- `NSH`:
  - Bit 27.
  - If set to `1`, will increment the associated `PMEVCNTR<n>` at EL2.
  - Reserved if EL2 not implemented.
- `SH`:
  - Bit 24.
  - If different to the `NSH` bit it enables the associated `PMEVCNTR<n>` at Secure EL2.
  - Reserved if Secure EL2 not implemented.
- `M`:
  - Bit 26.
  - If equal to the `P` bit it enables the associated `PMEVCNTR<n>` at EL3.
- `evtCount[15:10]`:

    - Extension to `evtCount[9:0]`. Reserved unless ARMv8.1-PMU implemented.

- `evtCount[9:0]`:
    - The event number that the associated `PMEVCNTR<n>` will count.

For `PMCNTENSET_EL0`/`PMCNTENSET`, the most important fields are:

- `P[30:0]`:
    - Setting bit `P[n]` to `1` enables counter `PMEVCNTR<n>`.
    - The effects of `PMEVTYPER<n>` are applied on top of this. In other words, the counter will not increment at any privilege level or security state unless it is enabled here.
- `C`:
    - Bit 31.
    - If set to `1` enables the cycle counter `PMCCNTR`.

For `PMCR`/`PMCR_EL0`, the most important fields are:

- `DP`:
    - Bit 5.
    - If set to `1` it disables the cycle counter `PMCCNTR` where event counting (by `PMEVCNTR<n>`) is prohibited (e.g. EL2 and the Secure world).
    - If set to `0`, `PMCCNTR` will not be affected by this bit and therefore will be able to count where the programmable counters are prohibited.
- `E`:
    - Bit 0.
    - Enables/disables counting altogether.
    - The effects of `PMCNTENSET` and `PMCR.DP` are applied on top of this. In other words, if this bit is `0` then no counters will increment regardless of how the other PMU system registers or bit fields are configured.

References

- Arm ARM

CHAPTER

EIGHT

SECURITY ADVISORIES

8.1 Advisory TFV-1 (CVE-2016-10319)

Title	Malformed Firmware Update SMC can result in copy of unexpectedly large data into secure memory
CVE ID	CVE-2016-10319
Date	18 Oct 2016
Versions Affected	v1.2 and v1.3 (since commit 48bfb88)
Configurations Affected	Platforms that use AArch64 BL1 plus untrusted normal world firmware update code executing before BL31
Impact	Copy of unexpectedly large data into the free secure memory reported by BL1 platform code
Fix Version	Pull Request #783
Credit	IOActive

Generic Trusted Firmware (TF) BL1 code contains an SMC interface that is briefly available after cold reset to support the Firmware Update (FWU) feature (also known as recovery mode). This allows most FWU functionality to be implemented in the normal world, while retaining the essential image authentication functionality in BL1. When cold boot reaches the EL3 Runtime Software (for example, BL31 on AArch64 systems), the FWU SMC interface is replaced by the EL3 Runtime SMC interface. Platforms may choose how much of this FWU functionality to use, if any.

The BL1 FWU SMC handling code, currently only supported on AArch64, contains several vulnerabilities that may be exploited when *all* the following conditions apply:

1. Platform code uses TF BL1 with the `TRUSTED_BOARD_BOOT` build option enabled.
2. Platform code arranges for untrusted normal world FWU code to be executed in the cold boot path, before BL31 starts. Untrusted in this sense means code that is not in ROM or has not been authenticated or has otherwise been executed by an attacker.
3. Platform code copies the insecure pattern described below from the ARM platform version of `bl1_plat_mem_check()`.

The vulnerabilities consist of potential integer overflows in the input validation checks while handling the `FWU_SMC_IMAGE_COPY` SMC. The SMC implementation is designed to copy an image into secure memory for subsequent authentication, but the vulnerabilities may allow an attacker to copy unexpectedly large data into secure memory. Note that a separate vulnerability is required to leverage these vulnerabilities; for example a way to get the system to change its behaviour based on the unexpected secure memory contents.

Two of the vulnerabilities are in the function `bl1_fwu_image_copy()` in `bl1/bl1_fwu.c`. These are listed below, referring to the v1.3 tagged version of the code:

- Line 155:

```
/*
 * If last block is more than expected then
 * clip the block to the required image size.
 */
if (image_desc->copied_size + block_size >
     image_desc->image_info.image_size) {
    block_size = image_desc->image_info.image_size -
        image_desc->copied_size;
    WARN("BL1-FWU: Copy argument block_size > remaining image size."
        " Clipping block_size\n");
}

/* Make sure the image src/size is mapped. */
if (bl1_plat_mem_check(image_src, block_size, flags)) {
    WARN("BL1-FWU: Copy arguments source/size not mapped\n");
    return -ENOMEM;
}

INFO("BL1-FWU: Continuing image copy in blocks\n");

/* Copy image for given block size. */
base_addr += image_desc->copied_size;
image_desc->copied_size += block_size;
memcpy((void *)base_addr, (const void *)image_src, block_size);
...
```

This code fragment is executed when the image copy operation is performed in blocks over multiple SMCs. `block_size` is an SMC argument and therefore potentially controllable by an attacker. A very large value may result in an integer overflow in the 1st `if` statement, which would bypass the check, allowing an unclipped `block_size` to be passed into `bl1_plat_mem_check()`. If `bl1_plat_mem_check()` also passes, this may result in an unexpectedly large copy of data into secure memory.

- Line 206:

```
/* Make sure the image src/size is mapped. */
if (bl1_plat_mem_check(image_src, block_size, flags)) {
    WARN("BL1-FWU: Copy arguments source/size not mapped\n");
    return -ENOMEM;
}

/* Find out how much free trusted ram remains after BL1 load */
mem_layout = bl1_plat_sec_mem_layout();
if ((image_desc->image_info.image_base < mem_layout->free_base) ||
     (image_desc->image_info.image_base + image_size >
      mem_layout->free_base + mem_layout->free_size)) {
    WARN("BL1-FWU: Memory not available to copy\n");
    return -ENOMEM;
}

/* Update the image size. */
image_desc->image_info.image_size = image_size;

/* Copy image for given size. */
memcpy((void *)base_addr, (const void *)image_src, block_size);
...
```

This code fragment is executed during the 1st invocation of the image copy operation. Both `block_size` and `image_size` are SMC arguments. A very large value of `image_size` may result in an integer overflow in the 2nd `if` statement, which would bypass the check, allowing execution to proceed. If `bl1_plat_mem_check()` also passes, this may result in an unexpectedly large copy of data into secure memory.

If the platform's implementation of `bl1_plat_mem_check()` is correct then it may help prevent the above 2 vulnerabilities from being exploited. However, the ARM platform version of this function contains a similar vulnerability:

- Line 88 of `plat/arm/common/arm_bl1_fwu.c` in function of `bl1_plat_mem_check()`:

```
while (mmap[index].mem_size) {
    if ((mem_base >= mmap[index].mem_base) &&
        ((mem_base + mem_size)
        <= (mmap[index].mem_base +
        mmap[index].mem_size)))
        return 0;

    index++;
}
...
```

  This function checks that the passed memory region is within one of the regions mapped in by ARM platforms. Here, `mem_size` may be the `block_size` passed from `bl1_fwu_image_copy()`. A very large value of `mem_size` may result in an integer overflow and the function to incorrectly return success. Platforms that copy this insecure pattern will have the same vulnerability.

8.2 Advisory TFV-2 (CVE-2017-7564)

Title	Enabled secure self-hosted invasive debug interface can allow normal world to panic secure world
CVE ID	CVE-2017-7564
Date	02 Feb 2017
Versions Affected	All versions up to v1.3
Configurations Affected	All
Impact	Denial of Service (secure world panic)
Fix Version	15 Feb 2017 Pull Request #841
Credit	ARM

The `MDCR_EL3.SDD` bit controls AArch64 secure self-hosted invasive debug enablement. By default, the BL1 and BL31 images of the current version of ARM Trusted Firmware (TF) unconditionally assign this bit to `0` in the early entrypoint code, which enables debug exceptions from the secure world. This can be seen in the implementation of the `el3_arch_init_common` AArch64 macro . Given that TF does not currently contain support for this feature (for example, by saving and restoring the appropriate debug registers), this may allow a normal world attacker to induce a panic in the secure world.

The `MDCR_EL3.SDD` bit should be assigned to `1` to disable debug exceptions from the secure world.

Earlier versions of TF (prior to commit 495f3d3) did not assign this bit. Since the bit has an architecturally `UNKNOWN` reset value, earlier versions may or may not have the same problem, depending on the platform.

A similar issue applies to the `MDCR_EL3.SPD32` bits, which control AArch32 secure self-hosted invasive debug enablement. TF assigns these bits to `00` meaning that debug exceptions from Secure EL1 are enabled by the authentication interface. Therefore this issue only exists for AArch32 Secure EL1 code when secure privileged invasive debug is enabled by the authentication interface, at which point the device is vulnerable to other, more serious attacks anyway.

However, given that TF contains no support for handling debug exceptions, the `MDCR_EL3.SPD32` bits should be assigned to `10` to disable debug exceptions from AArch32 Secure EL1.

Finally, this also issue applies to AArch32 platforms that use the TF SP_MIN image or integrate the AArch32 equivalent of the `el3_arch_init_common` macro. Here the affected bits are `SDCR.SPD`, which should also be assigned to `10` instead of `00`

8.3 Advisory TFV-3 (CVE-2017-7563)

Title	RO memory is always executable at AArch64 Secure EL1
CVE ID	CVE-2017-7563
Date	06 Apr 2017
Versions Affected	v1.3 (since Pull Request #662)
Configurations Affected	AArch64 BL2, TSP or other users of xlat_tables library executing at AArch64 Secure EL1
Impact	Unexpected Privilege Escalation
Fix Version	Pull Request #924
Credit	ARM

The translation table library in ARM Trusted Firmware (TF) (under `lib/xlat_tables` and `lib/xlat_tables_v2`) provides APIs to help program translation tables in the MMU. The xlat_tables client specifies its required memory mappings in the form of `mmap_region` structures. Each `mmap_region` has memory attributes represented by the `mmap_attr_t` enumeration type. This contains flags to control data access permissions (`MT_RO`/`MT_RW`) and instruction execution permissions (`MT_EXECUTE`/`MT_EXECUTE_NEVER`). Thus a mapping specifying both `MT_RO` and `MT_EXECUTE_NEVER` should result in a Read-Only (RO), non-executable memory region.

This feature does not work correctly for AArch64 images executing at Secure EL1. Any memory region mapped as RO will always be executable, regardless of whether the client specified `MT_EXECUTE` or `MT_EXECUTE_NEVER`.

The vulnerability is known to affect the BL2 and Test Secure Payload (TSP) images on platforms that enable the `SEPARATE_CODE_AND_RODATA` build option, which includes all ARM standard platforms, and the upstream Xilinx and NVidia platforms. The RO data section for these images on these platforms is unexpectedly executable instead of non-executable. Other platforms or `xlat_tables` clients may also be affected.

The vulnerability primarily manifests itself after Pull Request #662. Before that, `xlat_tables` clients could not specify instruction execution permissions separately to data access permissions. All RO normal memory regions were implicitly executable. Before Pull Request #662. the vulnerability would only manifest itself for device memory mapped as RO; use of this mapping is considered rare, although the upstream QEMU platform uses this mapping when the `DEVICE2_BASE` build option is used.

Note that one or more separate vulnerabilities are also required to exploit this vulnerability.

The vulnerability is due to incorrect handling of the execute-never bits in the translation tables. The EL3 translation regime uses a single `XN` bit to determine whether a region is executable. The Secure EL1&0 translation regime handles 2 Virtual Address (VA) ranges and so uses 2 bits, `UXN` and `PXN`. The `xlat_tables` library only handles the `XN` bit, which maps to `UXN` in the Secure EL1&0 regime. As a result, this programs the Secure EL0 execution permissions but always leaves the memory as executable at Secure EL1.

The vulnerability is mitigated by the following factors:

- The xlat_tables library ensures that all Read-Write (RW) memory regions are non-executable by setting the `SCTLR_ELx.WXN` bit. This overrides any value of the `XN`, `UXN` or `PXN` bits in the translation tables. See the `enable_mmu()` function:

```
sctlr = read_sctlr_el##_el();                  \
sctlr |= SCTLR_WXN_BIT | SCTLR_M_BIT;          \
```

- AArch32 configurations are unaffected. Here the `XN` bit controls execution privileges of the currently executing translation regime, which is the desired behaviour.
- ARM TF EL3 code (for example BL1 and BL31) ensures that all non-secure memory mapped into the secure world is non-executable by setting the `SCR_EL3.SIF` bit. See the `el3_arch_init_common` macro in `el3_common_macros.S`.

8.4 Advisory TFV-4 (CVE-2017-9607)

Title	Malformed Firmware Update SMC can result in copy or authentication of unexpected data in secure memory in AArch32 state
CVE ID	CVE-2017-9607
Date	20 Jun 2017
Versions Affected	None (only between 22 May 2017 and 14 June 2017)
Configurations Affected	Platforms that use AArch32 BL1 plus untrusted normal world firmware update code executing before BL31
Impact	Copy or authentication of unexpected data in the secure memory
Fix Version	Pull Request #979 (merged on 14 June 2017)
Credit	ARM

The `include/lib/utils_def.h` header file provides the `check_uptr_overflow()` macro, which aims at detecting arithmetic overflows that may occur when computing the sum of a base pointer and an offset. This macro evaluates to 1 if the sum of the given base pointer and offset would result in a value large enough to wrap around, which may lead to unpredictable behaviour.

The macro code is at line 52, referring to the version of the code as of commit c396b73:

```
/*
 * Evaluates to 1 if (ptr + inc) overflows, 0 otherwise.
 * Both arguments must be unsigned pointer values (i.e. uintptr_t).
 */
#define check_uptr_overflow(ptr, inc)        \
    (((ptr) > UINTPTR_MAX - (inc)) ? 1 : 0)
```

This macro does not work correctly for AArch32 images. It fails to detect overflows when the sum of its two parameters fall into the `[2^32, 2^64 - 1]` range. Therefore, any AArch32 code relying on this macro to detect such integer overflows is actually not protected.

The buggy code has been present in ARM Trusted Firmware (TF) since Pull Request #678 was merged (on 18 August 2016). However, the upstream code was not vulnerable until Pull Request #939 was merged (on 22 May 2017), which introduced AArch32 support for the Trusted Board Boot (TBB) feature. Before then, the `check_uptr_overflow()` macro was not used in AArch32 code.

The vulnerability resides in the BL1 FWU SMC handling code and it may be exploited when *all* the following conditions apply:

- Platform code uses TF BL1 with the `TRUSTED_BOARD_BOOT` build option.

- Platform code uses the Firmware Update (FWU) code provided in `bl1/bl1_fwu.c`, which is part of the TBB support.
- TF BL1 is compiled with the `ARCH=aarch32` build option.

In this context, the AArch32 BL1 image might fail to detect potential integer overflows in the input validation checks while handling the `FWU_SMC_IMAGE_COPY` and `FWU_SMC_IMAGE_AUTH` SMCs.

The `FWU_SMC_IMAGE_COPY` SMC handler is designed to copy an image into secure memory for subsequent authentication. This is implemented by the `bl1_fwu_image_copy()` function, which has the following function prototype:

```
static int bl1_fwu_image_copy(unsigned int image_id,
                    uintptr_t image_src,
                    unsigned int block_size,
                    unsigned int image_size,
                    unsigned int flags)
```

`image_src` is an SMC argument and therefore potentially controllable by an attacker. A very large 32-bit value, for example `2^32 -1`, may result in the sum of `image_src` and `block_size` overflowing a 32-bit type, which `check_uptr_overflow()` will fail to detect. Depending on its implementation, the platform-specific function `bl1_plat_mem_check()` might get defeated by these unsanitized values and allow the following memory copy operation, that would wrap around. This may allow an attacker to copy unexpected data into secure memory if the memory is mapped in BL1's address space, or cause a fatal exception if it's not.

The `FWU_SMC_IMAGE_AUTH` SMC handler is designed to authenticate an image resident in secure memory. This is implemented by the `bl1_fwu_image_auth()` function, which has the following function prototype:

```
static int bl1_fwu_image_auth(unsigned int image_id,
                    uintptr_t image_src,
                    unsigned int image_size,
                    unsigned int flags)
```

Similarly, if an attacker has control over the `image_src` or `image_size` arguments through the SMC interface and injects high values whose sum overflows, they might defeat the `bl1_plat_mem_check()` function and make the authentication module read data outside of what's normally allowed by the platform code or crash the platform.

Note that in both cases, a separate vulnerability is required to leverage this vulnerability; for example a way to get the system to change its behaviour based on the unexpected secure memory accesses. Moreover, the normal world FWU code would need to be compromised in order to send a malformed FWU SMC that triggers an integer overflow.

The vulnerability is known to affect all ARM standard platforms when enabling the `TRUSTED_BOARD_BOOT` and `ARCH=aarch32` build options. Other platforms may also be affected if they fulfil the above conditions.

8.5 Advisory TFV-5 (CVE-2017-15031)

Title	Not initializing or saving/restoring `PMCR_EL0` can leak secure world timing information
CVE ID	CVE-2017-15031
Date	02 Oct 2017, updated on 04 Nov 2019
Versions Affected	All, up to and including v2.1
Configurations Affected	All
Impact	Leakage of sensitive secure world timing information
Fix Version	Pull Request #1127 (merged on 18 October 2017) Commit e290a8fcbc (merged on 23 August 2019) Commit c3e8b0be9b (merged on 27 September 2019)
Credit	Arm, Marek Bykowski

The `PMCR_EL0` (Performance Monitors Control Register) provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters. If the `PMCR_EL0.DP` bit is set to zero, the cycle counter (when enabled) counts during secure world execution, even when prohibited by the debug signals.

Since TF-A does not save and restore `PMCR_EL0` when switching between the normal and secure worlds, normal world code can set `PMCR_EL0.DP` to zero to cause leakage of secure world timing information. This register should be added to the list of saved/restored registers both when entering EL3 and also transitioning to S-EL1.

Furthermore, `PMCR_EL0.DP` has an architecturally `UNKNOWN` reset value. Since Arm TF does not initialize this register, it's possible that on at least some implementations, `PMCR_EL0.DP` is set to zero by default. This and other bits with an architecturally UNKNOWN reset value should be initialized to sensible default values in the secure context.

The same issue exists for the equivalent AArch32 register, `PMCR`, except that here `PMCR_EL0.DP` architecturally resets to zero.

NOTE: The original pull request referenced above only fixed the issue for S-EL1 whereas the EL3 was fixed in the later commits.

8.6 Advisory TFV-6 (CVE-2017-5753, CVE-2017-5715, CVE-2017-5754)

Title	Trusted Firmware-A exposure to speculative processor vulnerabilities using cache timing side-channels
CVE ID	CVE-2017-5753 / CVE-2017-5715 / CVE-2017-5754
Date	03 Jan 2018 (Updated 11 Jan, 18 Jan, 26 Jan, 30 Jan and 07 June 2018)
Versions Affected	All, up to and including v1.4
Configurations Affected	All
Impact	Leakage of secure world data to normal world
Fix Version	Pull Request #1214, Pull Request #1228, Pull Request #1240 and Pull Request #1405
Credit	Google / Arm

This security advisory describes the current understanding of the Trusted Firmware-A exposure to the speculative processor vulnerabilities identified by Google Project Zero. To understand the background and wider impact of these vulnerabilities on Arm systems, please refer to the Arm Processor Security Update.

8.6.1 Variant 1 (CVE-2017-5753)

At the time of writing, no vulnerable patterns have been observed in upstream TF code, therefore no workarounds have been applied or are planned.

8.6.2 Variant 2 (CVE-2017-5715)

Where possible on vulnerable CPUs, Arm recommends invalidating the branch predictor as early as possible on entry into the secure world, before any branch instruction is executed. There are a number of implementation defined ways to achieve this.

For Cortex-A57 and Cortex-A72 CPUs, the Pull Requests (PRs) in this advisory invalidate the branch predictor when entering EL3 by disabling and re-enabling the MMU.

For Cortex-A73 and Cortex-A75 CPUs, the PRs in this advisory invalidate the branch predictor when entering EL3 by temporarily dropping into AArch32 Secure-EL1 and executing the `BPIALL` instruction. This workaround is significantly more complex than the "MMU disable/enable" workaround. The latter is not effective at invalidating the branch predictor on Cortex-A73/Cortex-A75.

Note that if other privileged software, for example a Rich OS kernel, implements its own branch predictor invalidation during context switch by issuing an SMC (to execute firmware branch predictor invalidation), then there is a dependency on the PRs in this advisory being deployed in order for those workarounds to work. If that other privileged software is able to workaround the vulnerability locally (for example by implementing "MMU disable/enable" itself), there is no such dependency.

Pull Request #1240 and Pull Request #1405 optimise the earlier fixes by implementing a specified CVE-2017-5715 workaround SMC (`SMCCC_ARCH_WORKAROUND_1`) for use by normal world privileged software. This is more efficient than calling an arbitrary SMC (for example `PSCI_VERSION`). Details of `SMCCC_ARCH_WORKAROUND_1` can be found in the CVE-2017-5715 mitigation specification. The specification and implementation also enable the normal world to discover the presence of this firmware service.

On Juno R1 we measured the round trip latency for both the `PSCI_VERSION` and `SMCCC_ARCH_WORKAROUND_1` SMCs on Cortex-A57, using both the "MMU disable/enable" and "BPIALL at AArch32 Secure-EL1" workarounds described above. This includes the time spent in test code conforming to the SMC Calling Convention (SMCCC) from AArch64. For the `SMCCC_ARCH_WORKAROUND_1` cases, the test code uses SMCCC v1.1, which reduces the number of general purpose registers it needs to save/restore. Although the `BPIALL` instruction is not effective at invalidating the branch predictor on Cortex-A57, the drop into Secure-EL1 with MMU disabled that this workaround entails effectively does invalidate the branch predictor. Hence this is a reasonable comparison.

The results were as follows:

Test	Time (ns)
`PSCI_VERSION` baseline (without PRs in this advisory)	515
`PSCI_VERSION` baseline (with PRs in this advisory)	527
`PSCI_VERSION` with "MMU disable/enable"	930
`SMCCC_ARCH_WORKAROUND_1` with "MMU disable/enable"	386
`PSCI_VERSION` with "BPIALL at AArch32 Secure-EL1"	1276
`SMCCC_ARCH_WORKAROUND_1` with "BPIALL at AArch32 Secure-EL1"	770

Due to the high severity and wide applicability of this issue, the above workarounds are enabled by default (on vulnerable CPUs only), despite some performance and code size overhead. Platforms can choose to disable them at compile time if they do not require them. Pull Request #1240 disables the workarounds for unaffected upstream platforms.

For vulnerable AArch32-only CPUs (for example Cortex-A8, Cortex-A9 and Cortex-A17), the `BPIALL` instruction should be used as early as possible on entry into the secure world. For Cortex-A8, also set `ACTLR[6]` to 1 during early processor initialization. Note that the `BPIALL` instruction is not effective at invalidating the branch predictor on Cortex-A15. For that CPU, set `ACTLR[0]` to 1 during early processor initialization, and invalidate the branch predictor by performing an `ICIALLU` instruction.

On AArch32 EL3 systems, the monitor and secure-SVC code is typically tightly integrated, for example as part of a Trusted OS. Therefore any Variant 2 workaround should be provided by vendors of that software and is outside the scope of TF. However, an example implementation in the minimal AArch32 Secure Payload, `SP_MIN` is provided in Pull Request #1228.

Other Arm CPUs are not vulnerable to this or other variants. This includes Cortex-A76, Cortex-A53, Cortex-A55, Cortex-A32, Cortex-A7 and Cortex-A5.

For more information about non-Arm CPUs, please contact the CPU vendor.

8.6.3 Variant 3 (CVE-2017-5754)

This variant is only exploitable between Exception Levels within the same translation regime, for example between EL0 and EL1, therefore this variant cannot be used to access secure memory from the non-secure world, and is not applicable for TF. However, Secure Payloads (for example, Trusted OS) should provide mitigations on vulnerable CPUs to protect themselves from exploited Secure-EL0 applications.

The only Arm CPU vulnerable to this variant is Cortex-A75.

8.7 Advisory TFV-7 (CVE-2018-3639)

Title	Trusted Firmware-A exposure to cache speculation vulnerability Variant 4
CVE ID	CVE-2018-3639
Date	21 May 2018 (Updated 7 June 2018)
Versions Affected	All, up to and including v1.5
Configurations Affected	All
Impact	Leakage of secure world data to normal world
Fix Version	Pull Request #1392, Pull Request #1397
Credit	Google

This security advisory describes the current understanding of the Trusted Firmware-A (TF-A) exposure to Variant 4 of the cache speculation vulnerabilities identified by Google Project Zero. To understand the background and wider impact of these vulnerabilities on Arm systems, please refer to the Arm Processor Security Update.

At the time of writing, the TF-A project is not aware of a Variant 4 exploit that could be used against TF-A. It is likely to be very difficult to achieve an exploit against current standard configurations of TF-A, due to the limited interfaces into the secure world with attacker-controlled inputs. However, this is becoming increasingly difficult to guarantee with the introduction of complex new firmware interfaces, for example the Software Delegated Exception Interface (SDEI). Also, the TF-A project does not have visibility of all vendor-supplied interfaces. Therefore, the TF-A project takes a conservative approach by mitigating Variant 4 in hardware wherever possible during secure world execution. The mitigation is enabled by setting an implementation defined control bit to prevent the re-ordering of stores and loads.

For each affected CPU type, TF-A implements one of the two following mitigation approaches in Pull Request #1392 and Pull Request #1397. Both approaches have a system performance impact, which varies for each CPU type and use-case. The mitigation code is enabled by default, but can be disabled at compile time for platforms that are unaffected or where the risk is deemed low enough.

Arm CPUs not mentioned below are unaffected.

8.7.1 Static mitigation

For affected CPUs, this approach enables the mitigation during EL3 initialization, following every PE reset. No mechanism is provided to disable the mitigation at runtime.

This approach permanently mitigates the entire software stack and no additional mitigation code is required in other software components.

TF-A implements this approach for the following affected CPUs:

- Cortex-A57 and Cortex-A72, by setting bit 55 (Disable load pass store) of `CPUACTLR_EL1` (`S3_1_C15_C2_0`).
- Cortex-A73, by setting bit 3 of `S3_0_C15_C0_0` (not documented in the Technical Reference Manual (TRM)).
- Cortex-A75, by setting bit 35 (reserved in TRM) of `CPUACTLR_EL1` (`S3_0_C15_C1_0`).

8.7.2 Dynamic mitigation

For affected CPUs, this approach also enables the mitigation during EL3 initialization, following every PE reset. In addition, this approach implements `SMCCC_ARCH_WORKAROUND_2` in the Arm architectural range to allow callers at lower exception levels to temporarily disable the mitigation in their execution context, where the risk is deemed low enough. This approach enables mitigation on entry to EL3, and restores the mitigation state of the lower exception level on exit from EL3. For more information on this approach, see Firmware interfaces for mitigating cache speculation vulnerabilities.

This approach may be complemented by additional mitigation code in other software components, for example code that calls `SMCCC_ARCH_WORKAROUND_2`. However, even without any mitigation code in other software components, this approach will effectively permanently mitigate the entire software stack, since the default mitigation state for firmware-managed execution contexts is enabled.

Since the expectation in this approach is that more software executes with the mitigation disabled, this may result in better system performance than the static approach for some systems or use-cases. However, for other systems or use-cases, this performance saving may be outweighed by the additional overhead of `SMCCC_ARCH_WORKAROUND_2` calls and TF-A exception handling.

TF-A implements this approach for the following affected CPU:

- Cortex-A76, by setting and clearing bit 16 (reserved in TRM) of `CPUACTLR2_EL1` (`S3_0_C15_C1_1`).

8.8 Advisory TFV-8 (CVE-2018-19440)

Title	Not saving x0 to x3 registers can leak information from one Normal World SMC client to another
CVE ID	CVE-2018-19440
Date	27 Nov 2018
Versions Affected	All
Configurations Affected	Multiple normal world SMC clients calling into AArch64 BL31
Impact	Leakage of SMC return values from one normal world SMC client to another
Fix Version	Pull Request #1710
Credit	Secmation

When taking an exception to EL3, BL31 saves the CPU context. The aim is to restore it before returning into the lower exception level software that called into the firmware. However, for an SMC exception, the general purpose registers `x0` to `x3` are not part of the CPU context saved on the stack.

As per the SMC Calling Convention, up to 4 values may be returned to the caller in registers `x0` to `x3`. In TF-A, these return values are written into the CPU context, typically using one of the `SMC_RETx()` macros provided in the `include/lib/aarch64/smccc_helpers.h` header file.

Before returning to the caller, the `restore_gp_registers()` function is called. It restores the values of all general purpose registers taken from the CPU context stored on the stack. This includes registers `x0` to `x3`, as can be seen in the `lib/el3_runtime/aarch64/context.S` file at line 339 (referring to the version of the code as of commit c385955):

```
/*
 * This function restores all general purpose registers except x30 from the
 * CPU context. x30 register must be explicitly restored by the caller.
 */
func restore_gp_registers
    ldp x0, x1, [sp, #CTX_GPREGS_OFFSET + CTX_GPREG_X0]
    ldp x2, x3, [sp, #CTX_GPREGS_OFFSET + CTX_GPREG_X2]
```

In the case of an SMC handler that does not use all 4 return values, the remaining ones are left unchanged in the CPU context. As a result, `restore_gp_registers()` restores the stale values saved by a previous SMC request (or asynchronous exception to EL3) that used these return values.

In the presence of multiple normal world SMC clients, this behaviour might leak some of the return values from one client to another. For example, if a victim client first sends an SMC that returns 4 values, a malicious client may then send a second SMC expecting no return values (for example, a `SDEI_EVENT_COMPLETE` SMC) to get the 4 return values of the victim client.

In general, the responsibility for mitigating threats due to the presence of multiple normal world SMC clients lies with EL2 software. When present, EL2 software must trap SMC calls from EL1 software to ensure secure behaviour.

For this reason, TF-A does not save `x0` to `x3` in the CPU context on an SMC synchronous exception. It has behaved this way since the first version.

We can confirm that at least upstream KVM-based systems mitigate this threat, and are therefore unaffected by this issue. Other EL2 software should be audited to assess the impact of this threat.

EL2 software might find mitigating this threat somewhat onerous, because for all SMCs it would need to be aware of which return registers contain valid data, so it can sanitise any unused return registers. On the other hand, mitigating this in EL3 is relatively easy and cheap. Therefore, TF-A will now ensure that no information is leaked through registers `x0` to `x3`, by preserving the register state over the call.

Note that AArch32 TF-A is not affected by this issue. The SMC handling code in `SP_MIN` already saves all general purpose registers - including `r0` to `r3`, as can be seen in the `include/lib/aarch32/smccc_macros.S` file at line 19 (referring to the version of the code as of commit c385955):

```
/*
 * Macro to save the General purpose registers (r0 - r12), the banked
 * spsr, lr, sp registers and the `scr` register to the SMC context on entry
 * due a SMC call. The `lr` of the current mode (monitor) is expected to be
 * already saved. The `sp` must point to the `smc_ctx_t` to save to.
 * Additionally, also save the 'pmcr' register as this is updated whilst
 * executing in the secure world.
 */
    .macro smccc_save_gp_mode_regs
    /* Save r0 - r12 in the SMC context */
    stm sp, {r0-r12}
```

CHAPTER

NINE

DESIGN DOCUMENTS

9.1 TF-A CMake buildsystem

Author Balint Dobszay

Organization Arm Limited

Contact Balint Dobszay <balint.dobszay@arm.com>

Status Accepted

Table of Contents

9.1.1 Abstract

This document presents a proposal for a new buildsystem for TF-A using CMake, and as part of this a reusable CMake framework for embedded projects. For a summary about the proposal, please see the Phabricator wiki page. As mentioned there, the proposal consists of two phases. The subject of this document is the first phase only.

9.1.2 Introduction

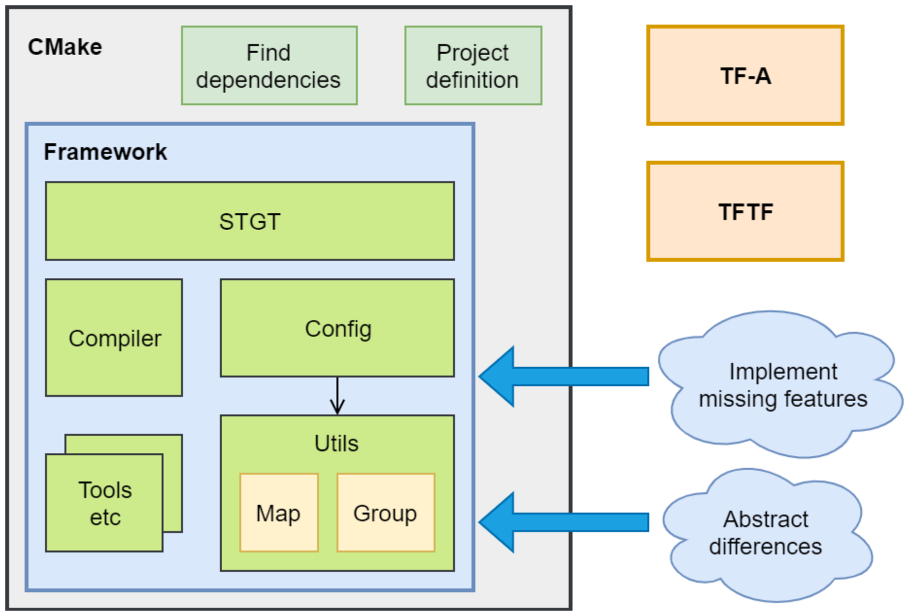
The current Makefile based buildsystem of TF-A has become complicated and hard to maintain, there is a need for a new, more flexible solution. The proposal is to use CMake language for the new buildsystem. The main reasons of this decision are the following:

- It is a well-established, mature tool, widely accepted by open-source projects.
- TF-M is already using CMake, reducing fragmentation for tf.org projects can be beneficial.
- CMake has various advantages over Make, e.g.:
  - Host and target system agnostic project.
  - CMake project is scalable, supports project modularization.
  - Supports software integration.
  - Out-of-the-box support for integration with several tools (e.g. project generation for various IDEs, integration with cppcheck, etc).

Of course there are drawbacks too:

- Language is problematic (e.g. variable scope).
- Not embedded approach.

To overcome these and other problems, we need to create workarounds for some tasks, wrap CMake functions, etc. Since this functionality can be useful in other embedded projects too, it is beneficial to collect the new code into a reusable framework and store this in a separate repository. The following diagram provides an overview of the framework structure:



9.1.3 Main features

Structured configuration description

In the current Makefile system the build configuration description, validation, processing, and the target creation, source file description are mixed and spread across several files. One of the goals of the framework is to organize this.

The framework provides a solution to describe the input build parameters, flags, macros, etc. in a structured way. It contains two utilities for this purpose:

- Map: simple key-value pair implementation.
- Group: collection of related maps.

The related parameters shall be packed into a group (or "setting group"). The setting groups shall be defined and filled with content in config files. Currently the config files are created and edited manually, but later a configuration management tool (e.g. Kconfig) shall be used to generate these files. Therefore, the framework does not contain parameter validation and conflict checking, these shall be handled by the configuration tool.

Target description

The framework provides an API called STGT ('simple target') to describe the targets, i.e. what is the build output, what source files are used, what libraries are linked, etc. The API wraps the CMake target functions, and also extends the built-in functionality, it can use the setting groups described in the previous section. A group can be applied onto a target, i.e. a collection of macros, flags, etc. can be applied onto the given output executable/library. This provides a more granular way than the current Makefile system where most of these are global and applied onto each target.

Compiler abstraction

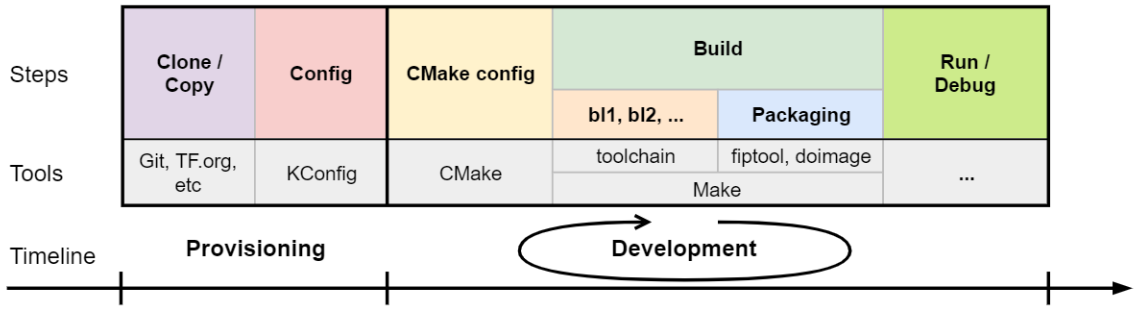
Apart from the built-in CMake usage of the compiler, there are some common tasks that CMake does not solve (e.g. preprocessing a file). For these tasks the framework uses wrapper functions instead of direct calls to the compiler. This way it is not tied to one specific compiler.

External tools

In the TF-A buildsystem some external tools are used, e.g. fiptool for image generation or dtc for device tree compilation. These tools have to be found and/or built by the framework. For this, the CMake find_package functionality is used, any other necessary tools can be added later.

9.1.4 Workflow

The following diagram demonstrates the development workflow using the framework:



The process can be split into two main phases:

In the provisioning phase, first we have to obtain the necessary resources, i.e. clone the code repository and other dependencies. Next we have to do the configuration, preferably using a config tool like KConfig.

In the development phase first we run CMake, which will generate the buildsystem using the selected generator backend (currently only the Makefile generator is supported). After this we run the selected build tool which in turn calls the compiler, linker, packaging tool, etc. Finally we can run and debug the output executables.

Usually during development only the steps in this second phase have to be repeated, while the provisioning phase needs to be done only once (or rarely).

9.1.5 Example

This is a short example for the basic framework usage.

First, we create a setting group called *mem_conf* and fill it with several parameters. It is worth noting the difference between *CONFIG* and *DEFINE* types: the former is only a CMake domain option, the latter is only a C language macro.

Next, we create a target called *fw1* and add the *mem_conf* setting group to it. This means that all source and header files used by the target will have all the parameters declared in the setting group. Then we set the target type to executable, and add some source files. Since the target has the parameters from the settings group, we can use it for conditionally adding source files. E.g. *dram_controller.c* will only be added if MEM_TYPE equals dram.

```
group_new(NAME mem_conf)
group_add(NAME mem_conf TYPE DEFINE KEY MEM_SIZE VAL 1024)
group_add(NAME mem_conf TYPE CONFIG DEFINE KEY MEM_TYPE VAL dram)
group_add(NAME mem_conf TYPE CFLAG KEY -Os)

stgt_create(NAME fw1)
stgt_add_setting(NAME fw1 GROUPS mem_conf)
stgt_set_target(NAME fw1 TYPE exe)

stgt_add_src(NAME fw1 SRC
    ${CMAKE_SOURCE_DIR}/main.c
)

stgt_add_src_cond(NAME fw1 KEY MEM_TYPE VAL dram SRC
    ${CMAKE_SOURCE_DIR}/dram_controller.c
)
```

CHAPTER

TEN

CHANGE LOG & RELEASE NOTES

This document contains a summary of the new features, changes, fixes and known issues in each release of Trusted Firmware-A.

10.1 Version 2.4

10.1.1 New Features

- **Architecture support**
  - **Armv8.6-A**
    - Added support for Armv8.6 Enhanced Counter Virtualization (ECV)
    - Added support for Armv8.6 Fine Grained Traps (FGT)
    - Added support for Armv8.6 WFE trap delays
- **Bootloader images**
  - Added support for Measured Boot
- **Build System**
  - Added build option `COT_DESC_IN_DTB` to create Chain of Trust at runtime
  - Added build option `OPENSSL_DIR` to direct tools to OpenSSL libraries
  - Added build option `RAS_TRAP_LOWER_EL_ERR_ACCESS` to enable trapping RAS register accesses from EL1/EL2 to EL3
  - Extended build option `BRANCH_PROTECTION` to support branch target identification
- **Common components**
  - Added support for exporting CPU nodes to the device tree
  - Added support for single and dual-root Chains of Trust in secure partitions
- **Drivers**
  - Added Broadcom RNG driver
  - Added Marvell `mg_conf_cm3` driver
  - Added System Control and Management Interface (SCMI) driver
  - Added STMicroelectronics ETZPC driver
  - **Arm GICv3**

    * Added support for detecting topology at runtime

- **Dual Root**
    * Added support for platform certificates
- **Marvell Cache LLC**
    * Added support for mapping the entire LLC into SRAM
- **Marvell CCU**
    * Added workaround for erratum 3033912
- **Marvell CP110 COMPHY**
    * Added support for SATA COMPHY polarity inversion
    * Added support for USB COMPHY polarity inversion
    * Added workaround for erratum IPCE_COMPHY-1353
- **STM32MP1 Clocks**
    * Added `RTC` as a gateable clock
    * Added support for shifted clock selector bit masks
    * Added support for using additional clocks as parents

- **Libraries**
  - **C standard library**
    * Added support for hexadecimal and pointer format specifiers in `snprint()`
    * Added assembly alternatives for various library functions
  - **CPU support**
    * **Arm Cortex-A53**
      · Added workaround for erratum 1530924
    * **Arm Cortex-A55**
      · Added workaround for erratum 1530923
    * **Arm Cortex-A57**
      · Added workaround for erratum 1319537
    * **Arm Cortex-A76**
      · Added workaround for erratum 1165522
      · Added workaround for erratum 1791580
      · Added workaround for erratum 1868343
    * **Arm Cortex-A72**
      · Added workaround for erratum 1319367
    * **Arm Cortex-A77**
      · Added workaround for erratum 1508412
      · Added workaround for erratum 1800714
      · Added workaround for erratum 1925769

        * **Arm Neoverse N1**
            · Added workaround for erratum 1868343
    - **EL3 Runtime**
        * Added support for saving/restoring registers related to nested virtualization in EL2 context switches if the architecture supports it
    - **FCONF**
        * Added support for Measured Boot
        * Added support for populating Chain of Trust properties
        * Added support for loading the `fw_config` image
    - **Measured Boot**
        * Added support for event logging

- **Platforms**
    - Added support for Arm Morello
    - Added support for Arm TC0
    - Added support for iEi PUZZLE-M801
    - Added support for Marvell OCTEON TX2 T9130
    - Added support for MediaTek MT8192
    - Added support for NXP i.MX 8M Nano
    - Added support for NXP i.MX 8M Plus
    - Added support for QTI CHIP SC7180
    - Added support for STM32MP151F
    - Added support for STM32MP153F
    - Added support for STM32MP157F
    - Added support for STM32MP151D
    - Added support for STM32MP153D
    - Added support for STM32MP157D
    - **Arm**
        * Added support for platform-owned SPs
        * Added support for resetting to BL31
    - **Arm FPGA**
        * Added support for Klein
        * Added support for Matterhorn
        * Added support for additional CPU clusters
    - **Arm FVP**
        * Added support for performing SDEI platform setup at runtime
        * Added support for SMCCC's `SMCCC_ARCH_SOC_ID` command

    * Added an `id` field under the NV-counter node in the device tree to differentiate between trusted and non-trusted NV-counters
    * Added support for extracting the clock frequency from the timer node in the device tree
- **Arm Juno**
    * Added support for SMCCC's `SMCCC_ARCH_SOC_ID` command
- **Arm N1SDP**
    * Added support for cross-chip PCI-e
- **Marvell**
    * Added support for AVS reduction
- **Marvell ARMADA**
    * Added support for twin-die combined memory device
- **Marvell ARMADA A8K**
    * Added support for DDR with 32-bit bus width (both ECC and non-ECC)
- **Marvell AP806**
    * Added workaround for erratum FE-4265711
- **Marvell AP807**
    * Added workaround for erratum 3033912
- **Nvidia Tegra**
    * Added debug printouts indicating SC7 entry sequence completion
    * Added support for SDEI
    * Added support for stack protection
    * Added support for GICv3
    * Added support for SMCCC's `SMCCC_ARCH_SOC_ID` command
- **Nvidia Tegra194**
    * Added support for RAS exception handling
    * Added support for SPM
- **NXP i.MX**
    * Added support for SDEI
- **QEMU SBSA**
    * Added support for the Secure Partition Manager
- **QTI**
    * Added RNG driver
    * Added SPMI PMIC arbitrator driver
    * Added support for SMCCC's `SMCCC_ARCH_SOC_ID` command
- **STM32MP1**
    * Added support for exposing peripheral interfaces to the non-secure world at runtime

        * Added support for SCMI clock and reset services
        * Added support for STM32MP15x CPU revision Z
        * Added support for SMCCC services in `SP_MIN`

- **Services**
    - **Secure Payload Dispatcher**
        * Added a provision to allow clients to retrieve the service UUID
    - **SPMC**
        * Added secondary core endpoint information to the SPMC context structure
    - **SPMD**
        * Added support for booting OP-TEE as a guest S-EL1 Secure Partition on top of Hafnium in S-EL2
        * Added a provision for handling SPMC messages to register secondary core entry points
        * Added support for power management operations
- **Tools**
    - **CertCreate**
        * Added support for secure partitions
    - **CertTool**
        * Added support for the `fw_config` image
    - **FIPTool**
        * Added support for the `fw_config` image

10.1.2 Changed

- Architecture support
- Bootloader images
- **Build System**
    - The top-level Makefile now supports building FipTool on Windows
    - The default value of `KEY_SIZE` has been changed to to 2048 when RSA is in use
    - The previously-deprecated macro `__ASSEMBLY__` has now been removed
- **Common components**
    - Certain functions that flush the console will no longer return error information
- **Drivers**
    - **Arm GIC**
        * Usage of `drivers/arm/gic/common/gic_common.c` has now been deprecated in favour of `drivers/arm/gic/vX/gicvX.mk`
        * Added support for detecting the presence of a GIC600-AE
        * Added support for detecting the presence of a GIC-Clayton
    - **Marvell MCI**

    * Now performs link tuning for all MCI interfaces to improve performance

  - **Marvell MoChi**
    * PIDI masters are no longer forced into a non-secure access level when `LLC_SRAM` is enabled
    * The SD/MMC controllers are now accessible from guest virtual machines
  - **Mbed TLS**
    * Migrated to Mbed TLS v2.24.0
  - **STM32 FMC2 NAND**
    * Adjusted FMC node bindings to include an EBI controller node
  - **STM32 Reset**
    * Added an optional timeout argument to assertion functions
  - **STM32MP1 Clocks**
    * Enabled several additional system clocks during initialization
- **Libraries**
  - **C Standard Library**
    * Improved `memset` performance by avoiding single-byte writes
    * Added optimized assembly variants of `memset`
  - **CPU support**
    * Renamed Cortex-Hercules to Cortex-A78
    * Renamed Cortex-Hercules AE to Cortex-A78 AE
    * Renamed Neoverse Zeus to Neoverse V1
  - **Coreboot**
    * Updated 'coreboot_get_memory_type' API to take an extra argument as a 'memory size' that used to return a valid memory type.
  - **libfdt**
    * Updated to latest upstream version
- **Platforms**
  - **Allwinner**
    * Disabled non-secure access to PRCM power control registers
  - **Arm**
    * `BL32_BASE` is now platform-dependent when `SPD_spmd` is enabled
    * Added support for loading the Chain of Trust from the device tree
    * The firmware update check is now executed only once
    * NV-counter base addresses are now loaded from the device tree when `COT_DESC_IN_DTB` is enabled
    * Now loads and populates `fw_config` and `tb_fw_config`
    * FCONF population now occurs after caches have been enabled in order to reduce boot times

- **Arm Corstone-700**
  - Platform support has been split into both an FVP and an FPGA variant
- **Arm FPGA**
  - DTB and BL33 load addresses have been given sensible default values
  - Now reads generic timer counter frequency, GICD and GICR base addresses, and UART address from DT
  - Now treats the primary PL011 UART as an SBSA Generic UART
- **Arm FVP**
  - Secure interrupt descriptions, UART parameters, clock frequencies and GICv3 parameters are now queried through FCONF
  - UART parameters are now queried through the device tree
  - Added an owner field to Cactus secure partitions
  - Increased the maximum size of BL2 when the Chain of Trust is loaded from the device tree
  - Reduces the maximum size of BL31
  - The `FVP_USE_SP804_TIMER` and `FVP_VE_USE_SP804_TIMER` build options have been removed in favour of a common `USE_SP804_TIMER` option
  - Added a third Cactus partition to manifests
  - Device tree nodes now store UUIDs in big-endian
- **Arm Juno**
  - Increased the maximum size of BL2 when optimizations have not been applied
  - Reduced the maximum size of BL31 and BL32
- **Marvell AP807**
  - Enabled snoop filters
- **Marvell ARMADA A3K**
  - UART recovery images are now suffixed with `.bin`
- **Marvell ARMADA A8K**
  - Option `BL31_CACHE_DISABLE` is now disabled (`0`) by default
- **Nvidia Tegra**
  - Added VPR resize supported check when processing video memory resize requests
  - Added SMMU verification to prevent potential issues caused by undetected corruption of the SMMU configuration during boot
  - The GIC CPU interface is now properly disabled after CPU off
  - The GICv2 sources list and the `BL31_SIZE` definition have been made platform-specific
  - The SPE driver will no longer flush the console when writing individual characters
- **Nvidia Tegra194**
  - TZDRAM setup has been moved to platform-specific early boot handlers
  - Increased verbosity of debug prints for RAS SErrors

    * Support for powering down CPUs during CPU suspend has been removed
    * Now verifies firewall settings before using resources

  - **TI K3**
    * The UART number has been made configurable through `K3_USART`

  - **Rockchip RK3368**
    * The maximum number of memory map regions has been increased to 20

  - **Socionext Uniphier**
    * The maximum size of BL33 has been increased to support larger bootloaders

  - **STM32**
    * Removed platform-specific DT functions in favour of using existing generic alternatives

  - **STM32MP1**
    * Increased verbosity of exception reports in debug builds
    * Device trees have been updated to align with the Linux kernel
    * Now uses the ETZPC driver to configure secure-aware interfaces for assignment to the non-secure world
    * Finished good variants have been added to the board identifier enumerations
    * Non-secure access to clocks and reset domains now depends on their state of registration
    * NEON is now disabled in `SP_MIN`
    * The last page of `SYSRAM` is now used as SCMI shared memory
    * Checks to verify platform compatibility have been added to verify that an image is compatible with the chip ID of the running platform

  - **QEMU SBSA**
    * Removed support for Arm's Cortex-A53

- **Services**
  - Renamed SPCI to FF-A
  - **SPMD**
    * No longer forwards requests to the non-secure world when retrieving partition information
    * SPMC manifest size is now retrieved directly from SPMD instead of the device tree
    * The FF-A version handler now returns SPMD's version when the origin of the call is secure, and SPMC's version when the origin of the call is non-secure
  - **SPMC**
    * Updated the manifest to declare CPU nodes in descending order as per the SPM (Hafnium) multicore requirement
    * Updated the device tree to mark 2GB as device memory for the first partition excluding trusted DRAM region (which is reserved for SPMC)
    * Increased the number of EC contexts to the maximum number of PEs as per the FF-A specification

- **Tools**

- **FIPTool**
  - Now returns `0` on `help` and `help <command>`
- **Marvell DoImage**
  - Updated Mbed TLS support to v2.8
- **SPTool**
  - Now appends CertTool arguments

10.1.3 Resolved Issues

- **Bootloader images**
  - Fixed compilation errors for dual-root Chains of Trust caused by symbol collision
  - **BL31**
    - Fixed compilation errors on platforms with fewer than 4 cores caused by initialization code exceeding the end of the stacks
    - Fixed compilation errors when building a position-independent image
- **Build System**
  - Fixed invalid empty version strings
  - Fixed compilation errors on Windows caused by a non-portable architecture revision comparison
- **Drivers**
  - **Arm GIC**
    - Fixed spurious interrupts caused by a missing barrier
  - **STM32 Flexible Memory Controller 2 (FMC2) NAND driver**
    - Fixed runtime instability caused by incorrect error detection logic
  - **STM32MP1 Clock driver**
    - Fixed incorrectly-formatted log messages
    - Fixed runtime instability caused by improper clock gating procedures
  - **STMicroelectronics Raw NAND driver**
    - Fixed runtime instability caused by incorrect unit conversion when waiting for NAND readiness
- **Libraries**
  - **AMU**
    - Fixed timeout errors caused by excess error logging
  - **EL3 Runtime**
    - Fixed runtime instability caused by improper register save/restore routine in EL2
  - **FCONF**
    - Fixed failure to initialize GICv3 caused by overly-strict device tree requirements
  - **Measured Boot**
    - Fixed driver errors caused by a missing default value for the `HASH_ALG` build option

- **SPE**
  - Fixed feature detection check that prevented CPUs supporting SVE from detecting support for SPE in the non-secure world
- **Translation Tables**
  - Fixed various MISRA-C 2012 static analysis violations

- **Platforms**
  - **Allwinner A64**
    - Fixed USB issues on certain battery-powered device caused by improperly activated USB power rail
  - **Arm**
    - Fixed compilation errors caused by increase in BL2 size
    - Fixed compilation errors caused by missing Makefile dependencies to generated files when building the FIP
    - Fixed MISRA-C 2012 static analysis violations caused by unused structures in include directives intended to be feature-gated
  - **Arm FPGA**
    - Fixed initialization issues caused by incorrect MPIDR topology mapping logic
  - **Arm RD-N1-edge**
    - Fixed compilation errors caused by mismatched parentheses in Makefile
  - **Arm SGI**
    - Fixed crashes due to the flash memory used for cold reboot attack protection not being mapped
  - **Intel Agilex**
    - Fixed initialization issues caused by several compounding bugs
  - **Marvell**
    - Fixed compilation warnings caused by multiple Makefile inclusions
  - **Marvell ARMADA A3K**
    - Fixed boot issue in debug builds caused by checks on the BL33 load address that are not appropriate for this platform
  - **Nvidia Tegra**
    - Fixed incorrect delay timer reads
    - Fixed spurious interrupts in the non-secure world during cold boot caused by the arbitration bit in the memory controller not being cleared
    - Fixed faulty video memory resize sequence
  - **Nvidia Tegra194**
    - Fixed incorrect alignment of TZDRAM base address
  - **NXP iMX8M**
    - Fixed CPU hot-plug issues caused by race condition

- **STM32MP1**
  - Fixed compilation errors in highly-parallel builds caused by incorrect Makefile dependencies
- **STM32MP157C-ED1**
  - Fixed initialization issues caused by missing device tree hash node
- **Raspberry Pi 3**
  - Fixed compilation errors caused by incorrect dependency ordering in Makefile
- **Rockchip**
  - Fixed initialization issues caused by non-critical errors when parsing FDT being treated as critical
- **Rockchip RK3368**
  - Fixed runtime instability caused by incorrect CPUID shift value
- **QEMU**
  - Fixed compilation errors caused by incorrect dependency ordering in Makefile
- **QEMU SBSA**
  - Fixed initialization issues caused by FDT exceeding reserved memory size
- **QTI**
  - Fixed compilation errors caused by inclusion of a non-existent file

- **Services**
  - **FF-A (previously SPCI)**
    - Fixed SPMD aborts caused by incorrect behaviour when the manifest is page-aligned
- **Tools**
  - Fixed compilation issues when compiling tools from within their respective directories
  - **FIPTool**
    - Fixed command line parsing issues on Windows when using arguments whose names also happen to be a subset of another's
  - **Marvell DoImage**
    - Fixed PKCS signature verification errors at boot on some platforms caused by generation of misaligned images

10.1.4 Known Issues

- **Platforms**
  - **NVIDIA Tegra**
    - Signed comparison compiler warnings occurring in libfdt are currently being worked around by disabling the warning for the platform until the underlying issue is resolved in libfdt

10.2 Version 2.3

10.2.1 New Features

- **Arm Architecture**
  - Add support for Armv8.4-SecEL2 extension through the SPCI defined SPMD/SPMC components.
  - Build option to support EL2 context save and restore in the secure world (CTX_INCLUDE_EL2_REGS).
  - Add support for SMCCC v1.2 (introducing the new SMCCC_ARCH_SOC_ID SMC). Note that the support is compliant, but the SVE registers save/restore will be done as part of future S-EL2/SPM development.
- **BL-specific**
  - Enhanced BL2 bootloader flow to load secure partitions based on firmware configuration data (fconf).
  - Changes necessary to support SEPARATE_NOBITS_REGION feature
  - TSP and BL2_AT_EL3: Add Position Independent Execution `PIE` support
- **Build System**
  - Add support for documentation build as a target in Makefile
  - Add `COT` build option to select the Chain of Trust to use when the Trusted Boot feature is enabled (default: `tbbr`).
  - Added creation and injection of secure partition packages into the FIP.
  - Build option to support SPMC component loading and run at S-EL1 or S-EL2 (SPMD_SPM_AT_SEL2).
  - Enable MTE support
  - Enable Link Time Optimization in GCC
  - Enable -Wredundant-decls warning check
  - Makefile: Add support to optionally encrypt BL31 and BL32
  - Add support to pass the nt_fw_config DTB to OP-TEE.
  - Introduce per-BL `CPPFLAGS`, `ASFLAGS`, and `LDFLAGS`
  - build_macros: Add CREATE_SEQ function to generate sequence of numbers
- **CPU Support**
  - cortex-a57: Enable higher performance non-cacheable load forwarding
  - Hercules: Workaround for Errata 1688305
  - Klein: Support added for Klein CPU
  - Matterhorn: Support added for Matterhorn CPU
- **Drivers**
  - auth: Add `calc_hash` function for hash calculation. Used for authentication of images when measured boot is enabled.

- cryptocell: Add authenticated decryption framework, and support for CryptoCell-713 and CryptoCell-712 RSA 3K
- gic600: Add support for multichip configuration and Clayton
- gicv3: Introduce makefile, Add extended PPI and SPI range, Add support for probing multiple GIC Redistributor frames
- gicv4: Add GICv4 extension for GIC driver
- io: Add an IO abstraction layer to load encrypted firmwares
- mhu: Derive doorbell base address
- mtd: Add SPI-NOR, SPI-NAND, SPI-MEM, and raw NAND framework
- scmi: Allow use of multiple SCMI channels
- scu: Add a driver for snoop control unit

• **Libraries**

- coreboot: Add memory range parsing and use generic base address
- compiler_rt: Import popcountdi2.c and popcountsi2.c files, aeabi_ldivmode.S file and dependencies
- debugFS: Add DebugFS functionality
- el3_runtime: Add support for enabling S-EL2
- fconf: Add Firmware Configuration Framework (fconf) (experimental).
- libc: Add memrchr function
- locks: bakery: Use is_dcache_enabled() helper and add a DMB to the 'read_cache_op' macro
- psci: Add support to enable different personality of the same soc.
- xlat_tables_v2: Add support to pass shareability attribute for normal memory region, use get_current_el_maybe_constant() in is_dcache_enabled(), read-only xlat tables for BL31 memory, and add enable_mmu()

• **New Platforms Support**

- arm/arm_fpga: New platform support added for FPGA
- arm/rddaniel: New platform support added for rd-daniel platform
- brcm/stingray: New platform support added for Broadcom stingray platform
- nvidia/tegra194: New platform support for Nvidia Tegra194 platform

• **Platforms**

- allwinner: Implement PSCI system suspend using SCPI, add a msgbox driver for use with SCPI, and reserve and map space for the SCP firmware
- allwinner: axp: Add AXP805 support
- allwinner: power: Add DLDO4 power rail
- amlogic: axg: Add a build flag when using ATOS as BL32 and support for the A113D (AXG) platform
- arm/a5ds: Add ethernet node and L2 cache node in devicetree
- arm/common: Add support for the new *dualroot* chain of trust
- arm/common: Add support for SEPARATE_NOBITS_REGION

- arm/common: Re-enable PIE when RESET_TO_BL31=1
- arm/common: Allow boards to specify second DRAM Base address and to define PLAT_ARM_TZC_FILTERS
- arm/cornstone700: Add support for mhuv2 and stack protector
- arm/fvp: Add support for fconf in BL31 and SP_MIN. Populate power domain desciptor dynamically by leveraging fconf APIs.
- arm/fvp: Add Cactus/Ivy Secure Partition information and use two instances of Cactus at S-EL1
- arm/fvp: Add support to run BL32 in TDRAM and BL31 in secure DRAM
- arm/fvp: Add support for GICv4 extension and BL2 hash calculation in BL1
- arm/n1sdp: Setup multichip gic routing table, update platform macros for dual-chip setup, introduce platform information SDS region, add support to update presence of External LLC, and enable the NEOVERSE_N1_EXTERNAL_LLC flag
- arm/rdn1edge: Add support for dual-chip configuration and use CREATE_SEQ helper macro to compare chip count
- arm/sgm: Always use SCMI for SGM platforms
- arm/sgm775: Add support for dynamic config using fconf
- arm/sgi: Add multi-chip mode parameter in HW_CONFIG dts, macros for remote chip device region, chip_id and multi_chip_mode to platform variant info, and introduce number of chips macro
- brcm: Add BL2 and BL31 support common across Broadcom platforms
- brcm: Add iproc SPI Nor flash support, spi driver, emmc driver, and support to retrieve plat_toc_flags
- hisilicon: hikey960: Enable system power off callback
- intel: Enable bridge access, SiP SMC secure register access, and uboot entrypoint support
- intel: Implement platform specific system reset 2
- intel: Introduce mailbox response length handling
- imx: console: Use CONSOLE_T_BASE for UART base address and generic console_t data structure
- imx8mm: Provide uart base as build option and add the support for opteed spd on imx8mq/imx8mm
- imx8qx: Provide debug uart num as build
- imx8qm: Apply clk/pinmux configuration for DEBUG_CONSOLE and provide debug uart num as build param
- marvell: a8k: Implement platform specific power off and add support for loading MG CM3 images
- mediatek: mt8183: Add Vmodem/Vcore DVS init level
- qemu: Support optional encryption of BL31 and BL32 images and ARM_LINUX_KERNEL_AS_BL33 to pass FDT address
- qemu: Define ARMV7_SUPPORTS_VFP
- qemu: Implement PSCI_CPU_OFF and qemu_system_off via semihosting
- renesas: rcar_gen3: Add new board revision for M3ULCB

  - rockchip: Enable workaround for erratum 855873, claim a macro to enable hdcp feature for DP, enable power domains of rk3399 before reset, add support for UART3 as serial output, and initialize reset and poweroff GPIOs with known invalid value
  - rpi: Implement PSCI CPU_OFF, use MMIO accessor, autodetect Mini-UART vs. PL011 configuration, and allow using PL011 UART for RPi3/RPi4
  - rpi3: Include GPIO driver in all BL stages and use same "clock-less" setup scheme as RPi4
  - rpi3/4: Add support for offlining CPUs
  - st: stm32mp1: platform.mk: Support generating multiple images in one build, migrate to implicit rules, derive map file name from target name, generate linker script with fixed name, and use PHONY for the appropriate targets
  - st: stm32mp1: Add support for SPI-NOR, raw NAND, and SPI-NAND boot device, QSPI, FMC2 driver
  - st: stm32mp1: Use stm32mp_get_ddr_ns_size() function, set XN attribute for some areas in BL2, dynamically map DDR later and non-cacheable during its test, add a function to get non-secure DDR size, add DT helper for reg by name, and add compilation flags for boot devices
  - socionext: uniphier: Turn on ENABLE_PIE
  - ti: k3: Add PIE support
  - xilinx: versal: Add set wakeup source, client wakeup, query data, request wakeup, PM_INIT_FINALIZE, PM_GET_TRUSTZONE_VERSION, PM IOCTL, support for suspend related, and Get_ChipID APIs
  - xilinx: versal: Implement power down/restart related EEMI, SMC handler for EEMI, PLL related PM, clock related PM, pin control related PM, reset related PM, device related PM , APIs
  - xilinx: versal: Enable ipi mailbox service
  - xilinx: versal: Add get_api_version support and support to send PM API to PMC using IPI
  - xilinx: zynqmp: Add checksum support for IPI data, GET_CALLBACK_DATA function, support to query max divisor, CLK_SET_RATE_PARENT in gem clock node, support for custom type flags, LPD WDT clock to the pm_clock structure, idcodes for new RFSoC silicons ZU48DR and ZU49DR, and id for new RFSoC device ZU39DR

- **Security**
  - Use Speculation Barrier instruction for v8.5+ cores
  - Add support for optional firmware encryption feature (experimental).
  - Introduce a new *dualroot* chain of trust.
  - aarch64: Prevent speculative execution past ERET
  - aarch32: Stop speculative execution past exception returns.
- **SPCI**
  - Introduced the Secure Partition Manager Dispatcher (SPMD) component as a new standard service.
- **Tools**
  - cert_create: Introduce CoT build option and TBBR CoT makefile, and define the dualroot CoT
  - encrypt_fw: Add firmware authenticated encryption tool
  - memory: Add show_memory script that prints a representation of the memory layout for the latest build

10.2.2 Changed

- **Arm Architecture**
  - PIE: Make call to GDT relocation fixup generalized
- **BL-Specific**
  - Increase maximum size of BL2 image
  - BL31: Discard .dynsym .dynstr .hash sections to make ENABLE_PIE work
  - BL31: Split into two separate memory regions
  - Unify BL linker scripts and reduce code duplication.
- **Build System**
  - Changes to drive cert_create for dualroot CoT
  - Enable -Wlogical-op always
  - Enable -Wshadow always
  - Refactor the warning flags
  - PIE: Pass PIE options only to BL31
  - Reduce space lost to object alignment
  - Set lld as the default linker for Clang builds
  - Remove -Wunused-const-variable and -Wpadded warning
  - Remove -Wmissing-declarations warning from WARNING1 level
- **Drivers**
  - authentication: Necessary fix in drivers to upgrade to mbedtls-2.18.0
  - console: Integrate UART base address in generic console_t
  - gicv3: Change API for GICR_IPRIORITYR accessors and separate GICD and GICR accessor functions
  - io: Change seek offset to signed long long and panic in case of io setup failure
  - smmu: SMMUv3: Changed retry loop to delay timer
  - tbbr: Reduce size of hash and ECDSA key buffers when possible
- **Library Code**
  - libc: Consolidate the size_t, unified, and NULL definitions, and unify intmax_t and uintmax_t on AArch32/64
  - ROMLIB: Optimize memory layout when ROMLIB is used
  - xlat_tables_v2: Use ARRAY_SIZE in REGISTER_XLAT_CONTEXT_FULL_SPEC, merge REGISTER_XLAT_CONTEXT_{FULL_SPEC,RO_BASE_TABLE}, and simplify end address checks in mmap_add_region_check()
- **Platforms**
  - allwinner: Adjust SRAM A2 base to include the ARISC vectors, clean up MMU setup, reenable USE_COHERENT_MEM, remove unused include path, move the NOBITS region to SRAM A1, convert AXP803 regulator setup code into a driver, enable clock before resetting I2C/RSB
  - allwinner: h6: power: Switch to using the AXP driver

- allwinner: a64: power: Use fdt_for_each_subnode, remove obsolete register check, remove duplicate DT check, and make sunxi_turn_off_soc static
- allwinner: Build PMIC bus drivers only in BL31, clean up PMIC-related error handling, and synchronize PMIC enumerations
- arm/a5ds: Change boot address to point to DDR address
- arm/common: Check for out-of-bound accesses in the platform io policies
- arm/corstone700: Updating the kernel arguments to support initramfs, use fdts DDR memory and XIP rootfs, and set UART clocks to 32MHz
- arm/fvp: Modify multithreaded dts file of DynamIQ FVPs, slightly bump the stack size for bl1 and bl2, remove re-definition of topology related build options, stop reclaiming init code with Clang builds, and map only the needed DRAM region statically in BL31/SP_MIN
- arm/juno: Maximize space allocated to SCP_BL2
- arm/sgi: Bump bl1 RW limit, mark remote chip shared ram as non-cacheable, move GIC related constants to board files, include AFF3 affinity in core position calculation, move bl31_platform_setup to board file, and move topology information to board folder
- common: Refactor load_auth_image_internal().
- hisilicon: Remove uefi-tools in hikey and hikey960 documentation
- intel: Modify non secure access function, BL31 address mapping, mailbox's get_config_status, and stratix10 BL31 parameter handling
- intel: Remove un-needed checks for qspi driver r/w and s10 unused source code
- intel: Change all global sip function to static
- intel: Refactor common platform code
- intel: Create SiP service header file
- marvell: armada: scp_bl2: Allow loading up to 8 images
- marvell: comphy-a3700: Support SGMII COMPHY power off and fix USB3 powering on when on lane 2
- marvell: Consolidate console register calls
- mediatek: mt8183: Protect 4GB~8GB dram memory, refine GIC driver for low power scenarios, and switch PLL/CLKSQ/ck_off/axi_26m control to SPM
- qemu: Update flash address map to keep FIP in secure FLASH0
- renesas: rcar_gen3: Update IPL and Secure Monitor Rev.2.0.6, update DDR setting for H3, M3, M3N, change fixed destination address of BL31 and BL32, add missing #{address,size}-cells into generated DT, pass DT to OpTee OS, and move DDR drivers out of staging
- rockchip: Make miniloader ddr_parameter handling optional, cleanup securing of ddr regions, move secure init to separate file, use base+size for secure ddr regions, bring TZRAM_SIZE values in lined, and prevent macro expansion in paths
- rpi: Move plat_helpers.S to common
- rpi3: gpio: Simplify GPIO setup
- rpi4: Skip UART initialisation
- st: stm32m1: Use generic console_t data structure, remove second QSPI flash instance, update for FMC2 pin muxing, and reduce MAX_XLAT_TABLES to 4

  - socionext: uniphier: Make on-chip SRAM and I/O register regions configurable
  - socionext: uniphier: Make PSCI related, counter control, UART, pinmon, NAND controller, and eMMC controller base addresses configurable
  - socionext: uniphier: Change block_addressing flag and the return value type of .is_usb_boot() to bool
  - socionext: uniphier: Run BL33 at EL2, call uniphier_scp_is_running() only when on-chip STM is supported, define PLAT_XLAT_TABLES_DYNAMIC only for BL2, support read-only xlat tables, use enable_mmu() in common function, shrink UNIPHIER_ROM_REGION_SIZE, prepare uniphier_soc_info() for next SoC, extend boot device detection for future SoCs, make all BL images completely position-independent, make uniphier_mmap_setup() work with PIE, pass SCP base address as a function parameter, set buffer offset and length for io_block dynamically, and use more mmap_add_dynamic_region() for loading images
  - spd/trusty: Disable error messages seen during boot, allow gic base to be specified with GICD_BASE, and allow getting trusty memsize from BL32_MEM_SIZE instead of TSP_SEC_MEM_SIZE
  - ti: k3: common: Enable ARM cluster power down and rename device IDs to be more consistent
  - ti: k3: drivers: ti_sci: Put sequence number in coherent memory and remove indirect structure of const data
  - xilinx: Move ipi mailbox svc to xilinx common
  - xilinx: zynqmp: Use GIC framework for warm restart
  - xilinx: zynqmp: pm: Move custom clock flags to typeflags, remove CLK_TOPSW_LSBUS from invalid clock list and rename FPD WDT clock ID
  - xilinx: versal: Increase OCM memory size for DEBUG builds and adjust cpu clock, Move versal_def.h and versal_private to include directory

- **Tools**
  - sptool: Updated sptool to accomodate building secure partition packages.

10.2.3 Resolved Issues

- **Arm Architecture**
  - Fix crash dump for lower EL
- **BL-Specific**
  - Bug fix: Protect TSP prints with lock
  - Fix boot failures on some builds linked with ld.lld.
- **Build System**
  - Fix clang build if CC is not in the path.
  - Fix 'BL stage' comment for build macros
- **Code Quality**
  - coverity: Fix various MISRA violations including null pointer violations, C issues in BL1/BL2/BL31 and FDT helper functions, using boolean essential, type, and removing unnecessary header file and comparisons to LONG_MAX in debugfs devfip

  - Based on coding guidelines, replace all *unsigned long* depending on if fixed based on AArch32 or AArch64.
  - Unify type of "cpu_idx" and Platform specific defines across PSCI module.

- **Drivers**
  - auth: Necessary fix in drivers to upgrade to mbedtls-2.18.0
  - delay_timer: Fix non-standard frequency issue in udelay
  - gicv3: Fix compiler dependent behavior
  - gic600: Fix include ordering according to the coding style and power up sequence
- **Library Code**
  - el3_runtime: Fix stack pointer maintenance on EA handling path, fixup 'cm_setup_context' prototype, and adds TPIDR_EL2 register to the context save restore routines
  - libc: Fix SIZE_MAX on AArch32
  - locks: T589: Fix insufficient ordering guarantees in bakery lock
  - pmf: Fix 'tautological-constant-compare' error, Make the runtime instrumentation work on AArch32, and Simplify PMF helper macro definitions across header files
  - xlat_tables_v2: Fix assembler warning of PLAT_RO_XLAT_TABLES
- **Platforms**
  - allwinner: Fix H6 GPIO and CCU memory map addresses and incorrect ARISC code patch offset check
  - arm/a5ds: Correct system freq and Cache Writeback Granule, and cleanup enable-method in device-tree
  - arm/fvp: Fix incorrect GIC mapping, BL31 load address and image size for RESET_TO_BL31=1, topology description of cpus for DynamIQ based FVP, and multithreaded FVP power domain tree
  - arm/fvp: spm-mm: Correcting instructions to build SPM for FVP
  - arm/common: Fix ROTPK hash generation for ECDSA encryption, BL2 bug in dynamic configuration initialisation, and current RECLAIM_INIT_CODE behavior
  - arm/rde1edge: Fix incorrect topology tree description
  - arm/sgi: Fix the incorrect check for SCMI channel ID
  - common: Flush dcache when storing timestamp
  - intel: Fix UEFI decompression issue, memory calibration, SMC SIP service, mailbox config return status, mailbox driver logic, FPGA manager on reconfiguration, and mailbox send_cmd issue
  - imx: Fix shift-overflow errors, the rdc memory region slot's offset, multiple definition of ipc_handle, missing inclusion of cdefs.h, and correct the SGIs that used for secure interrupt
  - mediatek: mt8183: Fix AARCH64 init fail on CPU0
  - rockchip: Fix definition of struct param_ddr_usage
  - rpi4: Fix documentation of armstub config entry
  - st: Correct io possible NULL pointer dereference and device_size type, nand xor_ecc.val assigned value, static analysis tool issues, and fix incorrect return value and correctly check pwr-regulators node
  - xilinx: zynqmp: Correct syscnt freq for QEMU and fix clock models and IDs of GEM-related clocks

10.2.4 Known Issues

- **Build System**
  - dtb: DTB creation not supported when building on a Windows host.

    This step in the build process is skipped when running on a Windows host. A known issue from the 1.6 release.
  - Intermittent assertion firing *ASSERT: services/spd/tspd/tspd_main.c:105*
- **Coverity**
  - Intermittent Race condition in Coverity Jenkins Build Job
- **Platforms**
  - arm/juno: System suspend from Linux does not function as documented in the user guide

    Following the instructions provided in the user guide document does not result in the platform entering system suspend state as expected. A message relating to the hdlcd driver failing to suspend will be emitted on the Linux terminal.
  - mediatek/mt6795: This platform does not build in this release

10.3 Version 2.2

10.3.1 New Features

- **Architecture**
  - **Enable Pointer Authentication (PAuth) support for Secure World**
    - Adds support for ARMv8.3-PAuth in BL1 SMC calls and BL2U image for firmware updates.
  - Enable Memory Tagging Extension (MTE) support in both secure and non-secure worlds
    - Adds support for the new Memory Tagging Extension arriving in ARMv8.5. MTE support is now enabled by default on systems that support it at EL0.
    - To enable it at ELx for both the non-secure and the secure world, the compiler flag `CTX_INCLUDE_MTE_REGS` includes register saving and restoring when necessary in order to prevent information leakage between the worlds.
  - Add support for Branch Target Identification (BTI)
- **Build System**
  - Modify FVP makefile for CPUs that support both AArch64/32
  - AArch32: Allow compiling with soft-float toolchain
  - Makefile: Add default warning flags
  - Add Makefile check for PAuth and AArch64
  - Add compile-time errors for HW_ASSISTED_COHERENCY flag
  - Apply compile-time check for AArch64-only CPUs
  - build_macros: Add mechanism to prevent bin generation.
  - Add support for default stack-protector flag

  - spd: opteed: Enable NS_TIMER_SWITCH
  - plat/arm: Skip BL2U if RESET_TO_SP_MIN flag is set
  - Add new build option to let each platform select which implementation of spinlocks it wants to use

- **CPU Support**
  - DSU: Workaround for erratum 798953 and 936184
  - Neoverse N1: Force cacheable atomic to near atomic
  - Neoverse N1: Workaround for erratum 1073348, 1130799, 1165347, 1207823, 1220197, 1257314, 1262606, 1262888, 1275112, 1315703, 1542419
  - Neoverse Zeus: Apply the MSR SSBS instruction
  - cortex-Hercules/HerculesAE: Support added for Cortex-Hercules and Cortex-HerculesAE CPUs
  - cortex-Hercules/HerculesAE: Enable AMU for Cortex-Hercules and Cortex-HerculesAE
  - cortex-a76AE: Support added for Cortex-A76AE CPU
  - cortex-a76: Workaround for erratum 1257314, 1262606, 1262888, 1275112, 1286807
  - cortex-a65/a65AE: Support added for Cortex-A65 and Cortex-A65AE CPUs
  - cortex-a65: Enable AMU for Cortex-A65
  - cortex-a55: Workaround for erratum 1221012
  - cortex-a35: Workaround for erratum 855472
  - cortex-a9: Workaround for erratum 794073
- **Drivers**
  - console: Allow the console to register multiple times
  - delay: Timeout detection support
  - gicv3: Enabled multi-socket GIC redistributor frame discovery and migrated ARM platforms to the new API
    * Adds `gicv3_rdistif_probe` function that delegates the responsibility of discovering the corresponding redistributor base frame to each CPU itself.
  - sbsa: Add SBSA watchdog driver
  - st/stm32_hash: Add HASH driver
  - ti/uart: Add an AArch32 variant
- **Library at ROM (romlib)**
  - Introduce BTI support in Library at ROM (romlib)
- **New Platforms Support**
  - amlogic: g12a: New platform support added for the S905X2 (G12A) platform
  - amlogic: meson/gxl: New platform support added for Amlogic Meson S905x (GXL)
  - arm/a5ds: New platform support added for A5 DesignStart
  - arm/corstone: New platform support added for Corstone-700
  - intel: New platform support added for Agilex
  - mediatek: New platform support added for MediaTek mt8183

- qemu/qemu_sbsa: New platform support added for QEMU SBSA platform
- renesas/rcar_gen3: plat: New platform support added for D3
- rockchip: New platform support added for px30
- rockchip: New platform support added for rk3288
- rpi: New platform support added for Raspberry Pi 4

- **Platforms**
  - arm/common: Introduce wrapper functions to setup secure watchdog
  - arm/fvp: Add Delay Timer driver to BL1 and BL31 and option for defining platform DRAM2 base
  - arm/fvp: Add Linux DTS files for 32 bit threaded FVPs
  - arm/n1sdp: Add code for DDR ECC enablement and BL33 copy to DDR, Initialise CNTFRQ in Non Secure CNTBaseN
  - arm/juno: Use shared mbedtls heap between BL1 and BL2 and add basic support for dynamic config
  - imx: Basic support for PicoPi iMX7D, rdc module init, caam module init, aipstz init, IMX_SIP_GET_SOC_INFO, IMX_SIP_BUILDINFO added
  - intel: Add ncore ccu driver
  - mediatek/mt81*: Use new bl31_params_parse() helper
  - nvidia: tegra: Add support for multi console interface
  - qemu/qemu_sbsa: Adding memory mapping for both FLASH0/FLASH1
  - qemu: Added gicv3 support, new console interface in AArch32, and sub-platforms
  - renesas/rcar_gen3: plat: Add R-Car V3M support, new board revision for H3ULCB, DBSC4 setting before self-refresh mode
  - socionext/uniphier: Support console based on multi-console
  - st: stm32mp1: Add OP-TEE, Avenger96, watchdog, LpDDR3, authentication support and general SYSCFG management
  - ti/k3: common: Add support for J721E, Use coherent memory for shared data, Trap all asynchronous bus errors to EL3
  - xilinx/zynqmp: Add support for multi console interface, Initialize IPI table from zynqmp_config_setup()

- **PSCI**
  - **Adding new optional PSCI hook `pwr_domain_on_finish_late`**
    * This PSCI hook `pwr_domain_on_finish_late` is similar to `pwr_domain_on_finish` but is guaranteed to be invoked when the respective core and cluster are participating in coherency.

- **Security**
  - Speculative Store Bypass Safe (SSBS): Further enhance protection against Spectre variant 4 by disabling speculative loads/stores (SPSR.SSBS bit) by default.
  - **UBSAN support and handlers**
    * Adds support for the Undefined Behaviour sanitizer. There are two types of support offered - minimalistic trapping support which essentially immediately crashes on undefined behaviour and full support with full debug messages.

- **Tools**
  - cert_create: Add support for bigger RSA key sizes (3KB and 4KB), previously the maximum size was 2KB.
  - fiptool: Add support to build fiptool on Windows.

10.3.2 Changed

- **Architecture**
  - Refactor ARMv8.3 Pointer Authentication support code
  - backtrace: Strip PAC field when PAUTH is enabled
  - Prettify crash reporting output on AArch64.
  - **Rework smc_unknown return code path in smc_handler**
    - Leverage the existing `el3_exit()` return routine for smc_unknown return path rather than a custom set of instructions.
- **BL-Specific**
  - Invalidate dcache build option for BL2 entry at EL3
  - Add missing support for BL2_AT_EL3 in XIP memory
- **Boot Flow**
  - Add helper to parse BL31 parameters (both versions)
  - Factor out cross-BL API into export headers suitable for 3rd party code
  - Introduce lightweight BL platform parameter library
- **Drivers**
  - auth: Memory optimization for Chain of Trust (CoT) description
  - bsec: Move bsec_mode_is_closed_device() service to platform
  - cryptocell: Move Cryptocell specific API into driver
  - gicv3: Prevent pending G1S interrupt from becoming G0 interrupt
  - mbedtls: Remove weak heap implementation
  - mmc: Increase delay between ACMD41 retries
  - mmc: stm32_sdmmc2: Correctly manage block size
  - mmc: stm32_sdmmc2: Manage max-frequency property from DT
  - synopsys/emmc: Do not change FIFO TH as this breaks some platforms
  - synopsys: Update synopsys drivers to not rely on undefined overflow behaviour
  - ufs: Extend the delay after reset to wait for some slower chips
- **Platforms**
  - amlogic/meson/gxl: Remove BL2 dependency from BL31
  - arm/common: Shorten the Firmware Update (FWU) process
  - arm/fvp: Remove GIC initialisation from secondary core cold boot
  - arm/sgm: Temporarily disable shared Mbed TLS heap for SGM

  - hisilicon: Update hisilicon drivers to not rely on undefined overflow behaviour
  - imx: imx8: Replace PLAT_IMX8* with PLAT_imx8*, remove duplicated linker symbols and deprecated code include, keep only IRQ 32 unmasked, enable all power domain by default
  - marvell: Prevent SError accessing PCIe link, Switch to xlat_tables_v2, do not rely on argument passed via smc, make sure that comphy init will use correct address
  - mediatek: mt8173: Refactor RTC and PMIC drivers
  - mediatek: mt8173: Apply MULTI_CONSOLE framework
  - nvidia: Tegra: memctrl_v2: fix "overflow before widen" coverity issue
  - qemu: Simplify the image size calculation, Move and generalise FDT PSCI fixup, move gicv2 codes to separate file
  - renesas/rcar_gen3: Convert to multi-console API, update QoS setting, Update IPL and Secure Monitor Rev2.0.4, Change to restore timer counter value at resume, Update DDR setting rev.0.35, qos: change subslot cycle, Change periodic write DQ training option.
  - rockchip: Allow SOCs with undefined wfe check bits, Streamline and complete UARTn_BASE macros, drop rockchip-specific imported linker symbols for bl31, Disable binary generation for all SOCs, Allow console device to be set by DTB, Use new bl31_params_parse functions
  - rpi/rpi3: Move shared rpi3 files into common directory
  - socionext/uniphier: Set CONSOLE_FLAG_TRANSLATE_CRLF and clean up console driver
  - socionext/uniphier: Replace DIV_ROUND_UP() with div_round_up() from utils_def.h
  - st/stm32mp: Split stm32mp_io_setup function, move stm32_get_gpio_bank_clock() to private file, correctly handle Clock Spreading Generator, move oscillator functions to generic file, realign device tree files with internal devs, enable RTCAPB clock for dual-core chips, use a common function to check spinlock is available, move check_header() to common code
  - ti/k3: Enable SEPARATE_CODE_AND_RODATA by default, Remove shared RAM space, Drop _ADDRESS from K3_USART_BASE to match other defines, Remove MSMC port definitions, Allow USE_COHERENT_MEM for K3, Set L2 latency on A72 cores

- **PSCI**
  - PSCI: Lookup list of parent nodes to lock only once
- **Secure Partition Manager (SPM): SPCI Prototype**
  - Fix service UUID lookup
  - Adjust size of virtual address space per partition
  - Refactor xlat context creation
  - Move shim layer to TTBR1_EL1
  - Ignore empty regions in resource description
- **Security**
  - Refactor SPSR initialisation code
  - **SMMUv3: Abort DMA transactions**
    * For security DMA should be blocked at the SMMU by default unless explicitly enabled for a device. SMMU is disabled after reset with all streams bypassing the SMMU, and abortion of all incoming transactions implements a default deny policy on reset.

        * Moves `bl1_platform_setup()` function from arm_bl1_setup.c to FVP platforms' fvp_bl1_setup.c and fvp_ve_bl1_setup.c files.

- **Tools**
  - cert_create: Remove RSA PKCS#1 v1.5 support

10.3.3 Resolved Issues

- **Architecture**
  - Fix the CAS spinlock implementation by adding a missing DSB in `spin_unlock()`
  - **AArch64: Fix SCTLR bit definitions**
    * Removes incorrect `SCTLR_V_BIT` definition and adds definitions for ARMv8.3-Pauth *EnIB*, *EnDA* and *EnDB* bits.
  - **Fix restoration of PAuth context**
    * Replace call to `pauth_context_save()` with `pauth_context_restore()` in case of unknown SMC call.
- **BL-Specific Issues**
  - Fix BL31 crash reporting on AArch64 only platforms
- **Build System**
  - Remove several warnings reported with W=2 and W=1
- **Code Quality Issues**
  - SCTLR and ACTLR are 32-bit for AArch32 and 64-bit for AArch64
  - Unify type of "cpu_idx" across PSCI module.
  - Assert if power level value greater then PSCI_INVALID_PWR_LVL
  - Unsigned long should not be used as per coding guidelines
  - Reduce the number of memory leaks in cert_create
  - Fix type of cot_desc_ptr
  - Use explicit-width data types in AAPCS parameter structs
  - Add python configuration for editorconfig
  - BL1: Fix type consistency
  - Enable -Wshift-overflow=2 to check for undefined shift behavior
  - Updated upstream platforms to not rely on undefined overflow behaviour
- **Coverity Quality Issues**
  - Remove GGC ignore -Warray-bounds
  - Fix Coverity #261967, Infinite loop
  - Fix Coverity #343017, Missing unlock
  - Fix Coverity #343008, Side affect in assertion
  - Fix Coverity #342970, Uninitialized scalar variable
- **CPU Support**

  - cortex-a12: Fix MIDR mask

- **Drivers**
  - console: Remove Arm console unregister on suspend
  - gicv3: Fix support for full SPI range
  - scmi: Fix wrong payload length
- **Library Code**
  - libc: Fix sparse warning for __assert()
  - libc: Fix memchr implementation
- **Platforms**
  - rpi: rpi3: Fix compilation error when stack protector is enabled
  - socionext/uniphier: Fix compilation fail for SPM support build config
  - st/stm32mp1: Fix TZC400 configuration against non-secure DDR
  - ti/k3: common: Fix RO data area size calculation
- **Security**
  - **AArch32: Disable Secure Cycle Counter**
    * Changes the implementation for disabling Secure Cycle Counter. For ARMv8.5 the counter gets disabled by setting `SDCR.SCCD` bit on CPU cold/warm boot. For the earlier architectures PMCR register is saved/restored on secure world entry/exit from/to Non-secure state, and cycle counting gets disabled by setting PMCR.DP bit.
  - **AArch64: Disable Secure Cycle Counter**
    * For ARMv8.5 the counter gets disabled by setting `MDCR_El3.SCCD` bit on CPU cold/warm boot. For the earlier architectures PMCR_EL0 register is saved/restored on secure world entry/exit from/to Non-secure state, and cycle counting gets disabled by setting PMCR_EL0.DP bit.

10.3.4 Deprecations

- **Common Code**
  - Remove MULTI_CONSOLE_API flag and references to it
  - Remove deprecated *plat_crash_console_**
  - Remove deprecated interfaces *get_afflvl_shift*, *mpidr_mask_lower_afflvls*, *eret*
  - AARCH32/AARCH64 macros are now deprecated in favor of `__aarch64__`
  - `__ASSEMBLY__` macro is now deprecated in favor of `__ASSEMBLER__`
- **Drivers**
  - console: Removed legacy console API
  - console: Remove deprecated finish_console_register
  - tzc: Remove deprecated types *tzc_action_t* and *tzc_region_attributes_t*
- **Secure Partition Manager (SPM):**

- Prototype SPCI-based SPM (services/std_svc/spm) will be replaced with alternative methods of secure partitioning support.

10.3.5 Known Issues

- **Build System Issues**
  - dtb: DTB creation not supported when building on a Windows host.

    This step in the build process is skipped when running on a Windows host. A known issue from the 1.6 release.
- **Platform Issues**
  - arm/juno: System suspend from Linux does not function as documented in the user guide

    Following the instructions provided in the user guide document does not result in the platform entering system suspend state as expected. A message relating to the hdlcd driver failing to suspend will be emitted on the Linux terminal.
  - mediatek/mt6795: This platform does not build in this release

10.4 Version 2.1

10.4.1 New Features

- **Architecture**
  - Support for ARMv8.3 pointer authentication in the normal and secure worlds

    The use of pointer authentication in the normal world is enabled whenever architectural support is available, without the need for additional build flags.

    Use of pointer authentication in the secure world remains an experimental configuration at this time. Using both the `ENABLE_PAUTH` and `CTX_INCLUDE_PAUTH_REGS` build flags, pointer authentication can be enabled in EL3 and S-EL1/0.

    See the *Firmware Design* document for additional details on the use of pointer authentication.
  - Enable Data Independent Timing (DIT) in EL3, where supported
- **Build System**
  - Support for BL-specific build flags
  - Support setting compiler target architecture based on `ARM_ARCH_MINOR` build option.
  - New `RECLAIM_INIT_CODE` build flag:

    A significant amount of the code used for the initialization of BL31 is not needed again after boot time. In order to reduce the runtime memory footprint, the memory used for this code can be reclaimed after initialization.

    Certain boot-time functions were marked with the `__init` attribute to enable this reclamation.
- **CPU Support**
  - cortex-a76: Workaround for erratum 1073348
  - cortex-a76: Workaround for erratum 1220197
  - cortex-a76: Workaround for erratum 1130799

- cortex-a75: Workaround for erratum 790748
- cortex-a75: Workaround for erratum 764081
- cortex-a73: Workaround for erratum 852427
- cortex-a73: Workaround for erratum 855423
- cortex-a57: Workaround for erratum 817169
- cortex-a57: Workaround for erratum 814670
- cortex-a55: Workaround for erratum 903758
- cortex-a55: Workaround for erratum 846532
- cortex-a55: Workaround for erratum 798797
- cortex-a55: Workaround for erratum 778703
- cortex-a55: Workaround for erratum 768277
- cortex-a53: Workaround for erratum 819472
- cortex-a53: Workaround for erratum 824069
- cortex-a53: Workaround for erratum 827319
- cortex-a17: Workaround for erratum 852423
- cortex-a17: Workaround for erratum 852421
- cortex-a15: Workaround for erratum 816470
- cortex-a15: Workaround for erratum 827671

• **Documentation**

- Exception Handling Framework documentation
- Library at ROM (romlib) documentation
- RAS framework documentation
- Coding Guidelines document

• **Drivers**

- **ccn: Add API for setting and reading node registers**
  - Adds `ccn_read_node_reg` function
  - Adds `ccn_write_node_reg` function
- partition: Support MBR partition entries
- scmi: Add `plat_css_get_scmi_info` function

  Adds a new API `plat_css_get_scmi_info` which lets the platform register a platform-specific instance of `scmi_channel_plat_info_t` and remove the default values
- tzc380: Add TZC-380 TrustZone Controller driver
- tzc-dmc620: Add driver to manage the TrustZone Controller within the DMC-620 Dynamic Memory Controller

• **Library at ROM (romlib)**

- Add platform-specific jump table list

- Allow patching of romlib functions

  This change allows patching of functions in the romlib. This can be done by adding "patch" at the end of the jump table entry for the function that needs to be patched in the file jmptbl.i.

- **Library Code**
  - Support non-LPAE-enabled MMU tables in AArch32
  - **mmio: Add `mmio_clrsetbits_16` function**
    - 16-bit variant of `mmio_clrsetbits`
  - **object_pool: Add Object Pool Allocator**
    - Manages object allocation using a fixed-size static array
    - Adds `pool_alloc` and `pool_alloc_n` functions
    - Does not provide any functions to free allocated objects (by design)
  - libc: Added `strlcpy` function
  - libc: Import `strrchr` function from FreeBSD
  - xlat_tables: Add support for ARMv8.4-TTST
  - xlat_tables: Support mapping regions without an explicitly specified VA
- **Math**
  - Added softudiv macro to support software division
- **Memory Partitioning And Monitoring (MPAM)**
  - Enabled MPAM EL2 traps (`MPAMHCR_EL2` and `MPAM_EL2`)
- **Platforms**
  - amlogic: Add support for Meson S905 (GXBB)
  - arm/fvp_ve: Add support for FVP Versatile Express platform
  - arm/n1sdp: Add support for Neoverse N1 System Development platform
  - arm/rde1edge: Add support for Neoverse E1 platform
  - arm/rdn1edge: Add support for Neoverse N1 platform
  - arm: Add support for booting directly to Linux without an intermediate loader (AArch32)
  - arm/juno: Enable new CPU errata workarounds for A53 and A57
  - arm/juno: Add romlib support

    Building a combined BL1 and ROMLIB binary file with the correct page alignment is now supported on the Juno platform. When `USE_ROMLIB` is set for Juno, it generates the combined file `bl1_romlib.bin` which needs to be used instead of bl1.bin.
  - intel/stratix: Add support for Intel Stratix 10 SoC FPGA platform
  - marvell: Add support for Armada-37xx SoC platform
  - nxp: Add support for i.MX8M and i.MX7 Warp7 platforms
  - renesas: Add support for R-Car Gen3 platform
  - xilinx: Add support for Versal ACAP platforms

- Position-Independent Executable (PIE)

  PIE support has initially been added to BL31. The `ENABLE_PIE` build flag is used to enable or disable this functionality as required.

- **Secure Partition Manager**
  - New SPM implementation based on SPCI Alpha 1 draft specification

    A new version of SPM has been implemented, based on the SPCI (Secure Partition Client Interface) and SPRT (Secure Partition Runtime) draft specifications.

    The new implementation is a prototype that is expected to undergo intensive rework as the specifications change. It has basic support for multiple Secure Partitions and Resource Descriptions.

    The older version of SPM, based on MM (ARM Management Mode Interface Specification), is still present in the codebase. A new build flag, `SPM_MM` has been added to allow selection of the desired implementation. This flag defaults to 1, selecting the MM-based implementation.

- **Security**
  - Spectre Variant-1 mitigations (`CVE-2017-5753`)
  - Use Speculation Store Bypass Safe (SSBS) functionality where available

    Provides mitigation against `CVE-2018-19440` (Not saving x0 to x3 registers can leak information from one Normal World SMC client to another)

10.4.2 Changed

- **Build System**
  - Warning levels are now selectable with `W=<1,2,3>`
  - Removed unneeded include paths in PLAT_INCLUDES
  - "Warnings as errors" (Werror) can be disabled using `E=0`
  - Support totally quiet output with `-s` flag
  - Support passing options to checkpatch using `CHECKPATCH_OPTS=<opts>`
  - Invoke host compiler with `HOSTCC / HOSTCCFLAGS` instead of `CC / CFLAGS`
  - **Make device tree pre-processing similar to U-boot/Linux by:**
    - Creating separate `CPPFLAGS` for DT preprocessing so that compiler options specific to it can be accommodated.
    - Replacing `CPP` with `PP` for DT pre-processing
- **CPU Support**
  - Errata report function definition is now mandatory for CPU support files

    CPU operation files must now define a `<name>_errata_report` function to print errata status. This is no longer a weak reference.

- **Documentation**
  - Migrated some content from GitHub wiki to `docs/` directory
  - Security advisories now have CVE links
  - Updated copyright guidelines
- **Drivers**

    - console: The `MULTI_CONSOLE_API` framework has been rewritten in C
    - console: Ported multi-console driver to AArch32
    - gic: Remove 'lowest priority' constants

      Removed `GIC_LOWEST_SEC_PRIORITY` and `GIC_LOWEST_NS_PRIORITY`. Platforms should define these if required, or instead determine the correct priority values at runtime.
    - delay_timer: Check that the Generic Timer extension is present
    - mmc: Increase command reply timeout to 10 milliseconds
    - mmc: Poll eMMC device status to ensure `EXT_CSD` command completion
    - mmc: Correctly check return code from `mmc_fill_device_info`

- External Libraries
  - libfdt: Upgraded from 1.4.2 to 1.4.6-9
  - mbed TLS: Upgraded from 2.12 to 2.16

    This change incorporates fixes for security issues that should be reviewed to determine if they are relevant for software implementations using Trusted Firmware-A. See the mbed TLS releases page for details on changes from the 2.12 to the 2.16 release.
- **Library Code**
    - compiler-rt: Updated `lshrdi3.c` and `int_lib.h` with changes from LLVM master branch (r345645)
    - cpu: Updated macro that checks need for `CVE-2017-5715` mitigation
    - libc: Made setjmp and longjmp C standard compliant
    - libc: Allowed overriding the default libc (use `OVERRIDE_LIBC`)
    - libc: Moved setjmp and longjmp to the `libc/` directory
- **Platforms**
    - Removed Mbed TLS dependency from plat_bl_common.c
    - arm: Removed unused `ARM_MAP_BL_ROMLIB` macro
    - arm: Removed `ARM_BOARD_OPTIMISE_MEM` feature and build flag
    - arm: Moved several components into `drivers/` directory

      This affects the SDS, SCP, SCPI, MHU and SCMI components
    - arm/juno: Increased maximum BL2 image size to `0xF000`

      This change was required to accommodate a larger `libfdt` library
- **SCMI**
    - Optimized bakery locks when hardware-assisted coherency is enabled using the `HW_ASSISTED_COHERENCY` build flag
- **SDEI**
    - Added support for unconditionally resuming secure world execution after *SDEI* event processing completes

      SDEI interrupts, although targeting EL3, occur on behalf of the non-secure world, and may have higher priority than secure world interrupts. Therefore they might preempt secure execution and yield execution to the non-secure *SDEI* handler. Upon completion of *SDEI* event handling, resume secure execution if it was preempted.

- **Translation Tables (XLAT)**
  - Dynamically detect need for `Common not Private (TTBRn_ELx.CnP)` bit

    Properly handle the case where `ARMv8.2-TTCNP` is implemented in a CPU that does not implement all mandatory v8.2 features (and so must claim to implement a lower architecture version).

10.4.3 Resolved Issues

- **Architecture**
  - Incorrect check for SSBS feature detection
  - Unintentional register clobber in AArch32 reset_handler function
- **Build System**
  - Dependency issue during DTB image build
  - Incorrect variable expansion in Arm platform makefiles
  - Building on Windows with verbose mode (`V=1`) enabled is broken
  - AArch32 compilation flags is missing `$(march32-directive)`
- **BL-Specific Issues**
  - bl2: `uintptr_t is not defined` error when `BL2_IN_XIP_MEM` is defined
  - bl2: Missing prototype warning in `bl2_arch_setup`
  - bl31: Omission of Global Offset Table (GOT) section
- **Code Quality Issues**
  - Multiple MISRA compliance issues
  - Potential NULL pointer dereference (Coverity-detected)
- **Drivers**
  - mmc: Local declaration of `scr` variable causes a cache issue when invalidating after the read DMA transfer completes
  - mmc: `ACMD41` does not send voltage information during initialization, resulting in the command being treated as a query. This prevents the command from initializing the controller.
  - mmc: When checking device state using `mmc_device_state()` there are no retries attempted in the event of an error
  - ccn: Incorrect Region ID calculation for RN-I nodes
  - console: `Fix MULTI_CONSOLE_API` when used as a crash console
  - partition: Improper NULL checking in gpt.c
  - partition: Compilation failure in `VERBOSE` mode (`V=1`)
- **Library Code**
  - common: Incorrect check for Address Authentication support

  - xlat: Fix XLAT_V1 / XLAT_V2 incompatibility

    The file `arm_xlat_tables.h` has been renamed to `xlat_tables_compat.h` and has been moved to a common folder. This header can be used to guarantee compatibility, as it includes the correct header based on `XLAT_TABLES_LIB_V2`.

  - xlat: armclang unused-function warning on `xlat_clean_dcache_range`
  - xlat: Invalid `mm_cursor` checks in `mmap_add` and `mmap_add_ctx`
  - sdei: Missing `context.h` header

- **Platforms**
  - common: Missing prototype warning for `plat_log_get_prefix`
  - arm: Insufficient maximum BL33 image size
  - arm: Potential memory corruption during BL2-BL31 transition

    On Arm platforms, the BL2 memory can be overlaid by BL31/BL32. The memory descriptors describing the list of executable images are created in BL2 R/W memory, which could be possibly corrupted later on by BL31/BL32 due to overlay. This patch creates a reserved location in SRAM for these descriptors and are copied over by BL2 before handing over to next BL image.

  - juno: Invalid behaviour when `CSS_USE_SCMI_SDS_DRIVER` is not set

    In `juno_pm.c` the `css_scmi_override_pm_ops` function was used regardless of whether the build flag was set. The original behaviour has been restored in the case where the build flag is not set.

- **Tools**
  - fiptool: Incorrect UUID parsing of blob parameters
  - doimage: Incorrect object rules in Makefile

10.4.4 Deprecations

- **Common Code**
  - `plat_crash_console_init` function
  - `plat_crash_console_putc` function
  - `plat_crash_console_flush` function
  - `finish_console_register` macro
- **AArch64-specific Code**
  - helpers: `get_afflvl_shift`
  - helpers: `mpidr_mask_lower_afflvls`
  - helpers: `eret`
- **Secure Partition Manager (SPM)**
  - Boot-info structure

10.4.5 Known Issues

- **Build System Issues**
  - dtb: DTB creation not supported when building on a Windows host.

    This step in the build process is skipped when running on a Windows host. A known issue from the 1.6 release.
- **Platform Issues**
  - arm/juno: System suspend from Linux does not function as documented in the user guide

    Following the instructions provided in the user guide document does not result in the platform entering system suspend state as expected. A message relating to the hdlcd driver failing to suspend will be emitted on the Linux terminal.
  - arm/juno: The firmware update use-cases do not work with motherboard firmware version < v1.5.0 (the reset reason is not preserved). The Linaro 18.04 release has MB v1.4.9. The MB v1.5.0 is available in Linaro 18.10 release.
  - mediatek/mt6795: This platform does not build in this release

10.5 Version 2.0

10.5.1 New Features

- Removal of a number of deprecated APIs
  - A new Platform Compatibility Policy document has been created which references a wiki page that maintains a listing of deprecated interfaces and the release after which they will be removed.
  - All deprecated interfaces except the MULTI_CONSOLE_API have been removed from the code base.
  - Various Arm and partner platforms have been updated to remove the use of removed APIs in this release.
  - This release is otherwise unchanged from 1.6 release

10.5.2 Issues resolved since last release

- No issues known at 1.6 release resolved in 2.0 release

10.5.3 Known Issues

- DTB creation not supported when building on a Windows host. This step in the build process is skipped when running on a Windows host. Known issue from 1.6 version.
- As a result of removal of deprecated interfaces the Nvidia Tegra, Marvell Armada 8K and MediaTek MT6795 platforms do not build in this release. Also MediaTek MT8173, NXP QorIQ LS1043A, NXP i.MX8QX, NXP i.MX8QMa, Rockchip RK3328, Rockchip RK3368 and Rockchip RK3399 platforms have not been confirmed to be working after the removal of the deprecated interfaces although they do build.

10.6 Version 1.6

10.6.1 New Features

- Addressing Speculation Security Vulnerabilities
  - Implement static workaround for CVE-2018-3639 for AArch32 and AArch64
  - Add support for dynamic mitigation for CVE-2018-3639
  - Implement dynamic mitigation for CVE-2018-3639 on Cortex-A76
  - Ensure *SDEI* handler executes with CVE-2018-3639 mitigation enabled
- Introduce RAS handling on AArch64
  - Some RAS extensions are mandatory for Armv8.2 CPUs, with others mandatory for Armv8.4 CPUs however, all extensions are also optional extensions to the base Armv8.0 architecture.
  - The Armv8 RAS Extensions introduced Standard Error Records which are a set of standard registers to configure RAS node policy and allow RAS Nodes to record and expose error information for error handling agents.
  - Capabilities are provided to support RAS Node enumeration and iteration along with individual interrupt registrations and fault injections support.
  - Introduce handlers for Uncontainable errors, Double Faults and EL3 External Aborts
- Enable Memory Partitioning And Monitoring (MPAM) for lower EL's
  - Memory Partitioning And Monitoring is an Armv8.4 feature that enables various memory system components and resources to define partitions. Software running at various ELs can then assign themselves to the desired partition to control their performance aspects.
  - When ENABLE_MPAM_FOR_LOWER_ELS is set to 1, EL3 allows lower ELs to access their own MPAM registers without trapping to EL3. This patch however, doesn't make use of partitioning in EL3; platform initialisation code should configure and use partitions in EL3 if required.
- Introduce ROM Lib Feature
  - Support combining several libraries into a self-called "romlib" image, that may be shared across images to reduce memory footprint. The romlib image is stored in ROM but is accessed through a jump-table that may be stored in read-write memory, allowing for the library code to be patched.
- Introduce Backtrace Feature
  - This function displays the backtrace, the current EL and security state to allow a post-processing tool to choose the right binary to interpret the dump.
  - Print backtrace in assert() and panic() to the console.
- Code hygiene changes and alignment with MISRA C-2012 guideline with fixes addressing issues complying to the following rules:
  - MISRA rules 4.9, 5.1, 5.3, 5.7, 8.2-8.5, 8.8, 8.13, 9.3, 10.1, 10.3-10.4, 10.8, 11.3, 11.6, 12.1, 14.4, 15.7, 16.1-16.7, 17.7-17.8, 20.7, 20.10, 20.12, 21.1, 21.15, 22.7
  - Clean up the usage of void pointers to access symbols
  - Increase usage of static qualifier to locally used functions and data
  - Migrated to use of u_register_t for register read/write to better match AArch32 and AArch64 type sizes
  - Use int-ll64 for both AArch32 and AArch64 to assist in consistent format strings between architectures

  - Clean up TF-A libc by removing non arm copyrighted implementations and replacing them with modified FreeBSD and SCC implementations
- Various changes to support Clang linker and assembler
  - The clang assembler/preprocessor is used when Clang is selected. However, the clang linker is not used because it is unable to link TF-A objects due to immaturity of clang linker functionality at this time.
- Refactor support APIs into Libraries
  - Evolve libfdt, mbed TLS library and standard C library sources as proper libraries that TF-A may be linked against.
- CPU Enhancements
  - Add CPU support for Cortex-Ares and Cortex-A76
  - Add AMU support for Cortex-Ares
  - Add initial CPU support for Cortex-Deimos
  - Add initial CPU support for Cortex-Helios
  - Implement dynamic mitigation for CVE-2018-3639 on Cortex-A76
  - Implement Cortex-Ares erratum 1043202 workaround
  - Implement DSU erratum 936184 workaround
  - Check presence of fix for errata 843419 in Cortex-A53
  - Check presence of fix for errata 835769 in Cortex-A53
- Translation Tables Enhancements
  - The xlat v2 library has been refactored in order to be reused by different TF components at different EL's including the addition of EL2. Some refactoring to make the code more generic and less specific to TF, in order to reuse the library outside of this project.
- SPM Enhancements
  - General cleanups and refactoring to pave the way to multiple partitions support
- SDEI Enhancements
  - Allow platforms to define explicit events
  - Determine client EL from NS context's SCR_EL3
  - Make dispatches synchronous
  - Introduce jump primitives for BL31
  - Mask events after CPU wakeup in *SDEI* dispatcher to conform to the specification
- Misc TF-A Core Common Code Enhancements
  - Add support for eXecute In Place (XIP) memory in BL2
  - Add support for the SMC Calling Convention 2.0
  - Introduce External Abort handling on AArch64 External Abort routed to EL3 was reported as an unhandled exception and caused a panic. This change enables Trusted Firmware-A to handle External Aborts routed to EL3.
  - Save value of ACTLR_EL1 implementation-defined register in the CPU context structure rather than forcing it to 0.

   - Introduce ARM_LINUX_KERNEL_AS_BL33 build option, which allows BL31 to directly jump to a Linux kernel. This makes for a quicker and simpler boot flow, which might be useful in some test environments.
   - Add dynamic configurations for BL31, BL32 and BL33 enabling support for Chain of Trust (COT).
   - Make TF UUID RFC 4122 compliant
- New Platform Support
   - Arm SGI-575
   - Arm SGM-775
   - Allwinner sun50i_64
   - Allwinner sun50i_h6
   - NXP QorIQ LS1043A
   - NXP i.MX8QX
   - NXP i.MX8QM
   - NXP i.MX7Solo WaRP7
   - TI K3
   - Socionext Synquacer SC2A11
   - Marvell Armada 8K
   - STMicroelectronics STM32MP1
- Misc Generic Platform Common Code Enhancements
   - Add MMC framework that supports both eMMC and SD card devices
- Misc Arm Platform Common Code Enhancements
   - Demonstrate PSCI MEM_PROTECT from el3_runtime
   - Provide RAS support
   - Migrate AArch64 port to the multi console driver. The old API is deprecated and will eventually be removed.
   - Move BL31 below BL2 to enable BL2 overlay resulting in changes in the layout of BL images in memory to enable more efficient use of available space.
   - Add cpp build processing for dtb that allows processing device tree with external includes.
   - Extend FIP io driver to support multiple FIP devices
   - Add support for SCMI AP core configuration protocol v1.0
   - Use SCMI AP core protocol to set the warm boot entrypoint
   - Add support to Mbed TLS drivers for shared heap among different BL images to help optimise memory usage
   - Enable non-secure access to UART1 through a build option to support a serial debug port for debugger connection
- Enhancements for Arm Juno Platform
   - Add support for TrustZone Media Protection 1 (TZMP1)
- Enhancements for Arm FVP Platform

  - Dynamic_config: remove the FVP dtb files
  - Set DYNAMIC_WORKAROUND_CVE_2018_3639=1 on FVP by default
  - Set the ability to dynamically disable Trusted Boot Board authentication to be off by default with DYN_DISABLE_AUTH
  - Add librom enhancement support in FVP
  - Support shared Mbed TLS heap between BL1 and BL2 that allow a reduction in BL2 size for FVP
- Enhancements for Arm SGI/SGM Platform
  - Enable ARM_PLAT_MT flag for SGI-575
  - Add dts files to enable support for dynamic config
  - Add RAS support
  - Support shared Mbed TLS heap for SGI and SGM between BL1 and BL2
- Enhancements for Non Arm Platforms
  - Raspberry Pi Platform
  - Hikey Platforms
  - Xilinx Platforms
  - QEMU Platform
  - Rockchip rk3399 Platform
  - TI Platforms
  - Socionext Platforms
  - Allwinner Platforms
  - NXP Platforms
  - NVIDIA Tegra Platform
  - Marvell Platforms
  - STMicroelectronics STM32MP1 Platform

10.6.2 Issues resolved since last release

- No issues known at 1.5 release resolved in 1.6 release

10.6.3 Known Issues

- DTB creation not supported when building on a Windows host. This step in the build process is skipped when running on a Windows host. Known issue from 1.5 version.

10.7 Version 1.5

10.7.1 New features

- Added new firmware support to enable RAS (Reliability, Availability, and Serviceability) functionality.
  - Secure Partition Manager (SPM): A Secure Partition is a software execution environment instantiated in S-EL0 that can be used to implement simple management and security services. The SPM is the firmware component that is responsible for managing a Secure Partition.
  - SDEI dispatcher: Support for interrupt-based *SDEI* events and all interfaces as defined by the *SDEI* specification v1.0, see SDEI Specification
  - Exception Handling Framework (EHF): Framework that allows dispatching of EL3 interrupts to their registered handlers which are registered based on their priorities. Facilitates firmware-first error handling policy where asynchronous exceptions may be routed to EL3.

    Integrated the TSPD with EHF.
- Updated PSCI support:
  - Implemented PSCI v1.1 optional features *MEM_PROTECT* and *SYSTEM_RESET2*. The supported PSCI version was updated to v1.1.
  - Improved PSCI STAT timestamp collection, including moving accounting for retention states to be inside the locks and fixing handling of wrap-around when calculating residency in AArch32 execution state.
  - Added optional handler for early suspend that executes when suspending to a power-down state and with data caches enabled.

    This may provide a performance improvement on platforms where it is safe to perform some or all of the platform actions from *pwr_domain_suspend* with the data caches enabled.
- Enabled build option, BL2_AT_EL3, for BL2 to allow execution at EL3 without any dependency on TF BL1.

  This allows platforms which already have a non-TF Boot ROM to directly load and execute BL2 and subsequent BL stages without need for BL1. This was not previously possible because BL2 executes at S-EL1 and cannot jump straight to EL3.
- Implemented support for SMCCC v1.1, including *SMCCC_VERSION* and *SMCCC_ARCH_FEATURES*.

  Additionally, added support for *SMCCC_VERSION* in PSCI features to enable discovery of the SMCCC version via PSCI feature call.
- Added Dynamic Configuration framework which enables each of the boot loader stages to be dynamically configured at runtime if required by the platform. The boot loader stage may optionally specify a firmware configuration file and/or hardware configuration file that can then be shared with the next boot loader stage.

  Introduced a new BL handover interface that essentially allows passing of 4 arguments between the different BL stages.

  Updated cert_create and fip_tool to support the dynamic configuration files. The COT also updated to support these new files.
- Code hygiene changes and alignment with MISRA guideline:
  - Fix use of undefined macros.
  - Achieved compliance with Mandatory MISRA coding rules.
  - Achieved compliance for following Required MISRA rules for the default build configurations on FVP and Juno platforms : 7.3, 8.3, 8.4, 8.5 and 8.8.

- Added support for Armv8.2-A architectural features:
  - Updated translation table set-up to set the CnP (Common not Private) bit for secure page tables so that multiple PEs in the same Inner Shareable domain can use the same translation table entries for a given stage of translation in a particular translation regime.
  - Extended the supported values of ID_AA64MMFR0_EL1.PARange to include the 52-bit Physical Address range.
  - Added support for the Scalable Vector Extension to allow Normal world software to access SVE functionality but disable access to SVE, SIMD and floating point functionality from the Secure world in order to prevent corruption of the Z-registers.
- **Added support for Armv8.4-A architectural feature Activity Monitor Unit (AMU)** extensions.

  In addition to the v8.4 architectural extension, AMU support on Cortex-A75 was implemented.
- Enhanced OP-TEE support to enable use of pageable OP-TEE image. The Arm standard platforms are updated to load up to 3 images for OP-TEE; header, pager image and paged image.

  The chain of trust is extended to support the additional images.
- Enhancements to the translation table library:
  - Introduced APIs to get and set the memory attributes of a region.
  - Added support to manage both privilege levels in translation regimes that describe translations for 2 Exception levels, specifically the EL1&0 translation regime, and extended the memory map region attributes to include specifying Non-privileged access.
  - Added support to specify the granularity of the mappings of each region, for instance a 2MB region can be specified to be mapped with 4KB page tables instead of a 2MB block.
  - Disabled the higher VA range to avoid unpredictable behaviour if there is an attempt to access addresses in the higher VA range.
  - Added helpers for Device and Normal memory MAIR encodings that align with the Arm Architecture Reference Manual for Armv8-A (Arm DDI0487B.b).
  - Code hygiene including fixing type length and signedness of constants, refactoring of function to enable the MMU, removing all instances where the virtual address space is hardcoded and added comments that document alignment needed between memory attributes and attributes specified in TCR_ELx.
- Updated GIC support:
  - Introduce new APIs for GICv2 and GICv3 that provide the capability to specify interrupt properties rather than list of interrupt numbers alone. The Arm platforms and other upstream platforms are migrated to use interrupt properties.
  - Added helpers to save / restore the GICv3 context, specifically the Distributor and Redistributor contexts and architectural parts of the ITS power management. The Distributor and Redistributor helpers also support the implementation-defined part of GIC-500 and GIC-600.

    Updated the Arm FVP platform to save / restore the GICv3 context on system suspend / resume as an example of how to use the helpers.

    Introduced a new TZC secured DDR carve-out for use by Arm platforms for storing EL3 runtime data such as the GICv3 register context.
- Added support for Armv7-A architecture via build option ARM_ARCH_MAJOR=7. This includes following features:
  - Updates GICv2 driver to manage GICv1 with security extensions.
  - Software implementation for 32bit division.

- Enabled use of generic timer for platforms that do not set ARM_CORTEX_Ax=yes.
- Support for Armv7-A Virtualization extensions [DDI0406C_C].
- Support for both Armv7-A platforms that only have 32-bit addressing and Armv7-A platforms that support large page addressing.
- Included support for following Armv7 CPUs: Cortex-A12, Cortex-A17, Cortex-A7, Cortex-A5, Cortex-A9, Cortex-A15.
- Added support in QEMU for Armv7-A/Cortex-A15.

- Enhancements to Firmware Update feature:
  - Updated the FWU documentation to describe the additional images needed for Firmware update, and how they are used for both the Juno platform and the Arm FVP platforms.
- Enhancements to Trusted Board Boot feature:
  - Added support to cert_create tool for RSA PKCS1# v1.5 and SHA384, SHA512 and SHA256.
  - For Arm platforms added support to use ECDSA keys.
  - Enhanced the mbed TLS wrapper layer to include support for both RSA and ECDSA to enable runtime selection between RSA and ECDSA keys.
- Added support for secure interrupt handling in AArch32 sp_min, hardcoded to only handle FIQs.
- Added support to allow a platform to load images from multiple boot sources, for example from a second flash drive.
- Added a logging framework that allows platforms to reduce the logging level at runtime and additionally the prefix string can be defined by the platform.
- Further improvements to register initialisation:
  - Control register PMCR_EL0 / PMCR is set to prohibit cycle counting in the secure world. This register is added to the list of registers that are saved and restored during world switch.
  - When EL3 is running in AArch32 execution state, the Non-secure version of SCTLR is explicitly initialised during the warmboot flow rather than relying on the hardware to set the correct reset values.
- Enhanced support for Arm platforms:
  - Introduced driver for Shared-Data-Structure (SDS) framework which is used for communication between SCP and the AP CPU, replacing Boot-Over_MHU (BOM) protocol.

    The Juno platform is migrated to use SDS with the SCMI support added in v1.3 and is set as default.

    The driver can be found in the plat/arm/css/drivers folder.
  - Improved memory usage by only mapping TSP memory region when the TSPD has been included in the build. This reduces the memory footprint and avoids unnecessary memory being mapped.
  - Updated support for multi-threading CPUs for FVP platforms - always check the MT field in MPDIR and access the bit fields accordingly.
  - Support building for platforms that model DynamIQ configuration by implementing all CPUs in a single cluster.
  - Improved nor flash driver, for instance clearing status registers before sending commands. Driver can be found plat/arm/board/common folder.
- Enhancements to QEMU platform:
  - Added support for TBB.

  - Added support for using OP-TEE pageable image.
  - Added support for LOAD_IMAGE_V2.
  - Migrated to use translation table library v2 by default.
  - Added support for SEPARATE_CODE_AND_RODATA.
- Applied workarounds CVE-2017-5715 on Arm Cortex-A57, -A72, -A73 and -A75, and for Armv7-A CPUs Cortex-A9, -A15 and -A17.
- Applied errata workaround for Arm Cortex-A57: 859972.
- Applied errata workaround for Arm Cortex-A72: 859971.
- Added support for Poplar 96Board platform.
- Added support for Raspberry Pi 3 platform.
- Added Call Frame Information (CFI) assembler directives to the vector entries which enables debuggers to display the backtrace of functions that triggered a synchronous abort.
- Added ability to build dtb.
- Added support for pre-tool (cert_create and fiptool) image processing enabling compression of the image files before processing by cert_create and fiptool.

  This can reduce fip size and may also speed up loading of images. The image verification will also get faster because certificates are generated based on compressed images.

  Imported zlib 1.2.11 to implement gunzip() for data compression.
- Enhancements to fiptool:
  - Enabled the fiptool to be built using Visual Studio.
  - Added padding bytes at the end of the last image in the fip to be facilitate transfer by DMA.

10.7.2 Issues resolved since last release

- TF-A can be built with optimisations disabled (-O0).
- Memory layout updated to enable Trusted Board Boot on Juno platform when running TF-A in AArch32 execution mode (resolving tf-issue#501).

10.7.3 Known Issues

- DTB creation not supported when building on a Windows host. This step in the build process is skipped when running on a Windows host.

10.8 Version 1.4

10.8.1 New features

- Enabled support for platforms with hardware assisted coherency.

  A new build option HW_ASSISTED_COHERENCY allows platforms to take advantage of the following optimisations:
  - Skip performing cache maintenance during power-up and power-down.

  - Use spin-locks instead of bakery locks.
  - Enable data caches early on warm-booted CPUs.

- Added support for Cortex-A75 and Cortex-A55 processors.

  Both Cortex-A75 and Cortex-A55 processors use the Arm DynamIQ Shared Unit (DSU). The power-down and power-up sequences are therefore mostly managed in hardware, reducing complexity of the software operations.

- Introduced Arm GIC-600 driver.

  Arm GIC-600 IP complies with Arm GICv3 architecture. For FVP platforms, the GIC-600 driver is chosen when FVP_USE_GIC_DRIVER is set to FVP_GIC600.

- Updated GICv3 support:

  - Introduced power management APIs for GICv3 Redistributor. These APIs allow platforms to power down the Redistributor during CPU power on/off. Requires the GICv3 implementations to have power management operations.

    Implemented the power management APIs for FVP.

  - GIC driver data is flushed by the primary CPU so that secondary CPU do not read stale GIC data.

- Added support for Arm System Control and Management Interface v1.0 (SCMI).

  The SCMI driver implements the power domain management and system power management protocol of the SCMI specification (Arm DEN 0056ASCMI) for communicating with any compliant power controller.

  Support is added for the Juno platform. The driver can be found in the plat/arm/css/drivers folder.

- Added support to enable pre-integration of TBB with the Arm TrustZone CryptoCell product, to take advantage of its hardware Root of Trust and crypto acceleration services.

- Enabled Statistical Profiling Extensions for lower ELs.

  The firmware support is limited to the use of SPE in the Non-secure state and accesses to the SPE specific registers from S-EL1 will trap to EL3.

  The SPE are architecturally specified for AArch64 only.

- Code hygiene changes aligned with MISRA guidelines:

  - Fixed signed / unsigned comparison warnings in the translation table library.
  - Added U(_x) macro and together with the existing ULL(_x) macro fixed some of the signed-ness defects flagged by the MISRA scanner.

- Enhancements to Firmware Update feature:

  - The FWU logic now checks for overlapping images to prevent execution of unauthenticated arbitrary code.
  - Introduced new FWU_SMC_IMAGE_RESET SMC that changes the image loading state machine to go from COPYING, COPIED or AUTHENTICATED states to RESET state. Previously, this was only possible when the authentication of an image failed or when the execution of the image finished.
  - Fixed integer overflow which addressed TFV-1: Malformed Firmware Update SMC can result in copy of unexpectedly large data into secure memory.

- Introduced support for Arm Compiler 6 and LLVM (clang).

  TF-A can now also be built with the Arm Compiler 6 or the clang compilers. The assembler and linker must be provided by the GNU toolchain.

  Tested with Arm CC 6.7 and clang 3.9.x and 4.0.x.

- Memory footprint improvements:
  - Introduced *tf_snprintf*, a reduced version of *snprintf* which has support for a limited set of formats.

    The mbedtls driver is updated to optionally use *tf_snprintf* instead of *snprintf*.
  - The *assert()* is updated to no longer print the function name, and additional logging options are supported via an optional platform define *PLAT_LOG_LEVEL_ASSERT*, which controls how verbose the assert output is.
- Enhancements to TF-A support when running in AArch32 execution state:
  - Support booting SP_MIN and BL33 in AArch32 execution mode on Juno. Due to hardware limitations, BL1 and BL2 boot in AArch64 state and there is additional trampoline code to warm reset into SP_MIN in AArch32 execution state.
  - Added support for Arm Cortex-A53/57/72 MPCore processors including the errata workarounds that are already implemented for AArch64 execution state.
  - For FVP platforms, added AArch32 Trusted Board Boot support, including the Firmware Update feature.
- Introduced Arm SiP service for use by Arm standard platforms.
  - Added new Arm SiP Service SMCs to enable the Non-secure world to read PMF timestamps.

    Added PMF instrumentation points in TF-A in order to quantify the overall time spent in the PSCI software implementation.
  - Added new Arm SiP service SMC to switch execution state.

    This allows the lower exception level to change its execution state from AArch64 to AArch32, or vice verse, via a request to EL3.
- Migrated to use SPDX[0] license identifiers to make software license auditing simpler.

  Note: Files that have been imported by FreeBSD have not been modified.

  [0]: https://spdx.org/
- Enhancements to the translation table library:
  - Added version 2 of translation table library that allows different translation tables to be modified by using different 'contexts'. Version 1 of the translation table library only allows the current EL's translation tables to be modified.

    Version 2 of the translation table also added support for dynamic regions; regions that can be added and removed dynamically whilst the MMU is enabled. Static regions can only be added or removed before the MMU is enabled.

    The dynamic mapping functionality is enabled or disabled when compiling by setting the build option PLAT_XLAT_TABLES_DYNAMIC to 1 or 0. This can be done per-image.
  - Added support for translation regimes with two virtual address spaces such as the one shared by EL1 and EL0.

    The library does not support initializing translation tables for EL0 software.
  - Added support to mark the translation tables as non-cacheable using an additional build option *XLAT_TABLE_NC*.
- Added support for GCC stack protection. A new build option ENABLE_STACK_PROTECTOR was introduced that enables compilation of all BL images with one of the GCC -fstack-protector-* options.

A new platform function plat_get_stack_protector_canary() was introduced that returns a value used to initialize the canary for stack corruption detection. For increased effectiveness of protection platforms must provide an implementation that returns a random value.

- Enhanced support for Arm platforms:
  - Added support for multi-threading CPUs, indicated by *MT* field in MPDIR. A new build flag *ARM_PLAT_MT* is added, and when enabled, the functions accessing MPIDR assume that the *MT* bit is set for the platform and access the bit fields accordingly.

    Also, a new API *plat_arm_get_cpu_pe_count* is added when *ARM_PLAT_MT* is enabled, returning the Processing Element count within the physical CPU corresponding to *mpidr*.
  - The Arm platforms migrated to use version 2 of the translation tables.
  - Introduced a new Arm platform layer API *plat_arm_psci_override_pm_ops* which allows Arm platforms to modify *plat_arm_psci_pm_ops* and therefore dynamically define PSCI capability.
  - The Arm platforms migrated to use IMAGE_LOAD_V2 by default.
- Enhanced reporting of errata workaround status with the following policy:
  - If an errata workaround is enabled:
    - If it applies (i.e. the CPU is affected by the errata), an INFO message is printed, confirming that the errata workaround has been applied.
    - If it does not apply, a VERBOSE message is printed, confirming that the errata workaround has been skipped.
  - If an errata workaround is not enabled, but would have applied had it been, a WARN message is printed, alerting that errata workaround is missing.
- Added build options ARM_ARCH_MAJOR and ARM_ARM_MINOR to choose the architecture version to target TF-A.
- Updated the spin lock implementation to use the more efficient CAS (Compare And Swap) instruction when available. This instruction was introduced in Armv8.1-A.
- Applied errata workaround for Arm Cortex-A53: 855873.
- Applied errata workaround for Arm-Cortex-A57: 813419.
- Enabled all A53 and A57 errata workarounds for Juno, both in AArch64 and AArch32 execution states.
- Added support for Socionext UniPhier SoC platform.
- Added support for Hikey960 and Hikey platforms.
- Added support for Rockchip RK3328 platform.
- Added support for NVidia Tegra T186 platform.
- Added support for Designware emmc driver.
- Imported libfdt v1.4.2 that addresses buffer overflow in fdt_offset_ptr().
- Enhanced the CPU operations framework to allow power handlers to be registered on per-level basis. This enables support for future CPUs that have multiple threads which might need powering down individually.
- Updated register initialisation to prevent unexpected behaviour:
  - Debug registers MDCR-EL3/SDCR and MDCR_EL2/HDCR are initialised to avoid unexpected traps into the higher exception levels and disable secure self-hosted debug. Additionally, secure privileged external debug on Juno is disabled by programming the appropriate Juno SoC registers.
  - EL2 and EL3 configurable controls are initialised to avoid unexpected traps in the higher exception levels.

  - Essential control registers are fully initialised on EL3 start-up, when initialising the non-secure and secure context structures and when preparing to leave EL3 for a lower EL. This gives better alignment with the Arm ARM which states that software must initialise RES0 and RES1 fields with 0 / 1.

- Enhanced PSCI support:

  - Introduced new platform interfaces that decouple PSCI stat residency calculation from PMF, enabling platforms to use alternative methods of capturing timestamps.
  - PSCI stat accounting performed for retention/standby states when requested at multiple power levels.

- Simplified fiptool to have a single linked list of image descriptors.
- For the TSP, resolved corruption of pre-empted secure context by aborting any pre-empted SMC during PSCI power management requests.

10.8.2 Issues resolved since last release

- TF-A can be built with the latest mbed TLS version (v2.4.2). The earlier version 2.3.0 cannot be used due to build warnings that the TF-A build system interprets as errors.
- TBBR, including the Firmware Update feature is now supported on FVP platforms when running TF-A in AArch32 state.
- The version of the AEMv8 Base FVP used in this release has resolved the issue of the model executing a reset instead of terminating in response to a shutdown request using the PSCI SYSTEM_OFF API.

10.8.3 Known Issues

- Building TF-A with compiler optimisations disabled (-O0) fails.
- Trusted Board Boot currently does not work on Juno when running Trusted Firmware in AArch32 execution state due to error when loading the sp_min to memory because of lack of free space available. See tf-issue#501 for more details.
- The errata workaround for A53 errata 843419 is only available from binutils 2.26 and is not present in GCC4.9. If this errata is applicable to the platform, please use GCC compiler version of at least 5.0. See PR#1002 for more details.

10.9 Version 1.3

10.9.1 New features

- Added support for running TF-A in AArch32 execution state.

  The PSCI library has been refactored to allow integration with **EL3 Runtime Software**. This is software that is executing at the highest secure privilege which is EL3 in AArch64 or Secure SVC/Monitor mode in AArch32. See *PSCI Library Integration guide for Armv8-A AArch32 systems*.

  Included is a minimal AArch32 Secure Payload, **SP-MIN**, that illustrates the usage and integration of the PSCI library with EL3 Runtime Software running in AArch32 state.

  Booting to the BL1/BL2 images as well as booting straight to the Secure Payload is supported.

- Improvements to the initialization framework for the PSCI service and Arm Standard Services in general.

  The PSCI service is now initialized as part of Arm Standard Service initialization. This consolidates the initializations of any Arm Standard Service that may be added in the future.

  A new function `get_arm_std_svc_args()` is introduced to get arguments corresponding to each standard service and must be implemented by the EL3 Runtime Software.

  For PSCI, a new versioned structure `psci_lib_args_t` is introduced to initialize the PSCI Library. **Note** this is a compatibility break due to the change in the prototype of `psci_setup()`.

- To support AArch32 builds of BL1 and BL2, implemented a new, alternative firmware image loading mechanism that adds flexibility.

  The current mechanism has a hard-coded set of images and execution order (BL31, BL32, etc). The new mechanism is data-driven by a list of image descriptors provided by the platform code.

  Arm platforms have been updated to support the new loading mechanism.

  The new mechanism is enabled by a build flag (`LOAD_IMAGE_V2`) which is currently off by default for the AArch64 build.

  Note `TRUSTED_BOARD_BOOT` is currently not supported when `LOAD_IMAGE_V2` is enabled.

- Updated requirements for making contributions to TF-A.

  Commits now must have a 'Signed-off-by:' field to certify that the contribution has been made under the terms of the `Developer Certificate of Origin`.

  A signed CLA is no longer required.

  The *Contributor's Guide* has been updated to reflect this change.

- Introduced Performance Measurement Framework (PMF) which provides support for capturing, storing, dumping and retrieving time-stamps to measure the execution time of critical paths in the firmware. This relies on defining fixed sample points at key places in the code.

- To support the QEMU platform port, imported libfdt v1.4.1 from https://git.kernel.org/pub/scm/utils/dtc/dtc.git

- Updated PSCI support:

  - Added support for PSCI NODE_HW_STATE API for Arm platforms.
  - New optional platform hook, `pwr_domain_pwr_down_wfi()`, in `plat_psci_ops` to enable platforms to perform platform-specific actions needed to enter powerdown, including the 'wfi' invocation.
  - PSCI STAT residency and count functions have been added on Arm platforms by using PMF.

- Enhancements to the translation table library:

  - Limited memory mapping support for region overlaps to only allow regions to overlap that are identity mapped or have the same virtual to physical address offset, and overlap completely but must not cover the same area.

    This limitation will enable future enhancements without having to support complex edge cases that may not be necessary.

  - The initial translation lookup level is now inferred from the virtual address space size. Previously, it was hard-coded.

  - Added support for mapping Normal, Inner Non-cacheable, Outer Non-cacheable memory in the translation table library.

    This can be useful to map a non-cacheable memory region, such as a DMA buffer.

  - Introduced the MT_EXECUTE/MT_EXECUTE_NEVER memory mapping attributes to specify the access permissions for instruction execution of a memory region.

- Enabled support to isolate code and read-only data on separate memory pages, allowing independent access control to be applied to each.
- Enabled SCR_EL3.SIF (Secure Instruction Fetch) bit in BL1 and BL31 common architectural setup code, preventing fetching instructions from non-secure memory when in secure state.
- Enhancements to FIP support:
  - Replaced `fip_create` with `fiptool` which provides a more consistent and intuitive interface as well as additional support to remove an image from a FIP file.
  - Enabled printing the SHA256 digest with info command, allowing quick verification of an image within a FIP without having to extract the image and running sha256sum on it.
  - Added support for unpacking the contents of an existing FIP file into the working directory.
  - Aligned command line options for specifying images to use same naming convention as specified by TBBR and already used in cert_create tool.
- Refactored the TZC-400 driver to also support memory controllers that integrate TZC functionality, for example Arm CoreLink DMC-500. Also added DMC-500 specific support.
- Implemented generic delay timer based on the system generic counter and migrated all platforms to use it.
- Enhanced support for Arm platforms:
  - Updated image loading support to make SCP images (SCP_BL2 and SCP_BL2U) optional.
  - Enhanced topology description support to allow multi-cluster topology definitions.
  - Added interconnect abstraction layer to help platform ports select the right interconnect driver, CCI or CCN, for the platform.
  - Added support to allow loading BL31 in the TZC-secured DRAM instead of the default secure SRAM.
  - Added support to use a System Security Control (SSC) Registers Unit enabling TF-A to be compiled to support multiple Arm platforms and then select one at runtime.
  - Restricted mapping of Trusted ROM in BL1 to what is actually needed by BL1 rather than entire Trusted ROM region.
  - Flash is now mapped as execute-never by default. This increases security by restricting the executable region to what is strictly needed.
- Applied following erratum workarounds for Cortex-A57: 833471, 826977, 829520, 828024 and 826974.
- Added support for Mediatek MT6795 platform.
- Added support for QEMU virtualization Armv8-A target.
- Added support for Rockchip RK3368 and RK3399 platforms.
- Added support for Xilinx Zynq UltraScale+ MPSoC platform.
- Added support for Arm Cortex-A73 MPCore Processor.
- Added support for Arm Cortex-A72 processor.
- Added support for Arm Cortex-A35 processor.
- Added support for Arm Cortex-A32 MPCore Processor.

- Enabled preloaded BL33 alternative boot flow, in which BL2 does not load BL33 from non-volatile storage and BL31 hands execution over to a preloaded BL33. The User Guide has been updated with an example of how to use this option with a bootwrapped kernel.
- Added support to build TF-A on a Windows-based host machine.
- Updated Trusted Board Boot prototype implementation:
  - Enabled the ability for a production ROM with TBBR enabled to boot test software before a real ROTPK is deployed (e.g. manufacturing mode). Added support to use ROTPK in certificate without verifying against the platform value when `ROTPK_NOT_DEPLOYED` bit is set.
  - Added support for non-volatile counter authentication to the Authentication Module to protect against roll-back.
- Updated GICv3 support:
  - Enabled processor power-down and automatic power-on using GICv3.
  - Enabled G1S or G0 interrupts to be configured independently.
  - Changed FVP default interrupt driver to be the GICv3-only driver. **Note** the default build of TF-A will not be able to boot Linux kernel with GICv2 FDT blob.
  - Enabled wake-up from CPU_SUSPEND to stand-by by temporarily re-routing interrupts and then restoring after resume.

10.9.2 Issues resolved since last release

10.9.3 Known issues

- The version of the AEMv8 Base FVP used in this release resets the model instead of terminating its execution in response to a shutdown request using the PSCI `SYSTEM_OFF` API. This issue will be fixed in a future version of the model.
- Building TF-A with compiler optimisations disabled (`-O0`) fails.
- TF-A cannot be built with mbed TLS version v2.3.0 due to build warnings that the TF-A build system interprets as errors.
- TBBR is not currently supported when running TF-A in AArch32 state.

10.10 Version 1.2

10.10.1 New features

- The Trusted Board Boot implementation on Arm platforms now conforms to the mandatory requirements of the TBBR specification.

  In particular, the boot process is now guarded by a Trusted Watchdog, which will reset the system in case of an authentication or loading error. On Arm platforms, a secure instance of Arm SP805 is used as the Trusted Watchdog.

  Also, a firmware update process has been implemented. It enables authenticated firmware to update firmware images from external interfaces to SoC Non-Volatile memories. This feature functions even when the current firmware in the system is corrupt or missing; it therefore may be used as a recovery mode.
- Improvements have been made to the Certificate Generation Tool (`cert_create`) as follows.

  - Added support for the Firmware Update process by extending the Chain of Trust definition in the tool to include the Firmware Update certificate and the required extensions.
  - Introduced a new API that allows one to specify command line options in the Chain of Trust description. This makes the declaration of the tool's arguments more flexible and easier to extend.
  - The tool has been reworked to follow a data driven approach, which makes it easier to maintain and extend.
- Extended the FIP tool (`fip_create`) to support the new set of images involved in the Firmware Update process.
- Various memory footprint improvements. In particular:
  - The bakery lock structure for coherent memory has been optimised.
  - The mbed TLS SHA1 functions are not needed, as SHA256 is used to generate the certificate signature. Therefore, they have been compiled out, reducing the memory footprint of BL1 and BL2 by approximately 6 KB.
  - On Arm development platforms, each BL stage now individually defines the number of regions that it needs to map in the MMU.
- Added the following new design documents:
  - *Authentication Framework & Chain of Trust*
  - *Firmware Update (FWU)*
  - *CPU Reset*
  - *PSCI Power Domain Tree Structure*
- Applied the new image terminology to the code base and documentation, as described in the *Image Terminology* document.
- The build system has been reworked to improve readability and facilitate adding future extensions.
- On Arm standard platforms, BL31 uses the boot console during cold boot but switches to the runtime console for any later logs at runtime. The TSP uses the runtime console for all output.
- Implemented a basic NOR flash driver for Arm platforms. It programs the device using CFI (Common Flash Interface) standard commands.
- Implemented support for booting EL3 payloads on Arm platforms, which reduces the complexity of developing EL3 baremetal code by doing essential baremetal initialization.
- Provided separate drivers for GICv3 and GICv2. These expect the entire software stack to use either GICv2 or GICv3; hybrid GIC software systems are no longer supported and the legacy Arm GIC driver has been deprecated.
- Added support for Juno r1 and r2. A single set of Juno TF-A binaries can run on Juno r0, r1 and r2 boards. Note that this TF-A version depends on a Linaro release that does *not* contain Juno r2 support.
- Added support for MediaTek mt8173 platform.
- Implemented a generic driver for Arm CCN IP.
- Major rework of the PSCI implementation.
  - Added framework to handle composite power states.
  - Decoupled the notions of affinity instances (which describes the hierarchical arrangement of cores) and of power domain topology, instead of assuming a one-to-one mapping.
  - Better alignment with version 1.0 of the PSCI specification.

- Added support for the SYSTEM_SUSPEND PSCI API on Arm platforms. When invoked on the last running core on a supported platform, this puts the system into a low power mode with memory retention.
- Unified the reset handling code as much as possible across BL stages. Also introduced some build options to enable optimization of the reset path on platforms that support it.
- Added a simple delay timer API, as well as an SP804 timer driver, which is enabled on FVP.
- Added support for NVidia Tegra T210 and T132 SoCs.
- Reorganised Arm platforms ports to greatly improve code shareability and facilitate the reuse of some of this code by other platforms.
- Added support for Arm Cortex-A72 processor in the CPU specific framework.
- Provided better error handling. Platform ports can now define their own error handling, for example to perform platform specific bookkeeping or post-error actions.
- Implemented a unified driver for Arm Cache Coherent Interconnects used for both CCI-400 & CCI-500 IPs. Arm platforms ports have been migrated to this common driver. The standalone CCI-400 driver has been deprecated.

10.10.2 Issues resolved since last release

- The Trusted Board Boot implementation has been redesigned to provide greater modularity and scalability. See the *Authentication Framework & Chain of Trust* document. All missing mandatory features are now implemented.
- The FVP and Juno ports may now use the hash of the ROTPK stored in the Trusted Key Storage registers to verify the ROTPK. Alternatively, a development public key hash embedded in the BL1 and BL2 binaries might be used instead. The location of the ROTPK is chosen at build-time using the `ARM_ROTPK_LOCATION` build option.
- GICv3 is now fully supported and stable.

10.10.3 Known issues

- The version of the AEMv8 Base FVP used in this release resets the model instead of terminating its execution in response to a shutdown request using the PSCI `SYSTEM_OFF` API. This issue will be fixed in a future version of the model.
- While this version has low on-chip RAM requirements, there are further RAM usage enhancements that could be made.
- The upstream documentation could be improved for structural consistency, clarity and completeness. In particular, the design documentation is incomplete for PSCI, the TSP(D) and the Juno platform.
- Building TF-A with compiler optimisations disabled (`-O0`) fails.

10.11 Version 1.1

10.11.1 New features

- A prototype implementation of Trusted Board Boot has been added. Boot loader images are verified by BL1 and BL2 during the cold boot path. BL1 and BL2 use the PolarSSL SSL library to verify certificates and images. The OpenSSL library is used to create the X.509 certificates. Support has been added to `fip_create` tool to package the certificates in a FIP.
- Support for calling CPU and platform specific reset handlers upon entry into BL3-1 during the cold and warm boot paths has been added. This happens after another Boot ROM `reset_handler()` has already run. This enables a developer to perform additional actions or undo actions already performed during the first call of the reset handlers e.g. apply additional errata workarounds.
- Support has been added to demonstrate routing of IRQs to EL3 instead of S-EL1 when execution is in secure world.
- The PSCI implementation now conforms to version 1.0 of the PSCI specification. All the mandatory APIs and selected optional APIs are supported. In particular, support for the `PSCI_FEATURES` API has been added. A capability variable is constructed during initialization by examining the `plat_pm_ops` and `spd_pm_ops` exported by the platform and the Secure Payload Dispatcher. This is used by the PSCI FEATURES function to determine which PSCI APIs are supported by the platform.
- Improvements have been made to the PSCI code as follows.
  - The code has been refactored to remove redundant parameters from internal functions.
  - Changes have been made to the code for PSCI `CPU_SUSPEND`, `CPU_ON` and `CPU_OFF` calls to facilitate an early return to the caller in case a failure condition is detected. For example, a PSCI `CPU_SUSPEND` call returns `SUCCESS` to the caller if a pending interrupt is detected early in the code path.
  - Optional platform APIs have been added to validate the `power_state` and `entrypoint` parameters early in PSCI `CPU_ON` and `CPU_SUSPEND` code paths.
  - PSCI migrate APIs have been reworked to invoke the SPD hook to determine the type of Trusted OS and the CPU it is resident on (if applicable). Also, during a PSCI `MIGRATE` call, the SPD hook to migrate the Trusted OS is invoked.
- It is now possible to build TF-A without marking at least an extra page of memory as coherent. The build flag `USE_COHERENT_MEM` can be used to choose between the two implementations. This has been made possible through these changes.
  - An implementation of Bakery locks, where the locks are not allocated in coherent memory has been added.
  - Memory which was previously marked as coherent is now kept coherent through the use of software cache maintenance operations.

  Approximately, 4K worth of memory is saved for each boot loader stage when `USE_COHERENT_MEM=0`. Enabling this option increases the latencies associated with acquire and release of locks. It also requires changes to the platform ports.
- It is now possible to specify the name of the FIP at build time by defining the `FIP_NAME` variable.
- Issues with dependencies on the 'fiptool' makefile target have been rectified. The `fip_create` tool is now rebuilt whenever its source files change.
- The BL3-1 runtime console is now also used as the crash console. The crash console is changed to SoC UART0 (UART2) from the previous FPGA UART0 (UART0) on Juno. In FVP, it is changed from UART0 to UART1.

- CPU errata workarounds are applied only when the revision and part number match. This behaviour has been made consistent across the debug and release builds. The debug build additionally prints a warning if a mismatch is detected.
- It is now possible to issue cache maintenance operations by set/way for a particular level of data cache. Levels 1-3 are currently supported.
- The following improvements have been made to the FVP port.
  - The build option `FVP_SHARED_DATA_LOCATION` which allowed relocation of shared data into the Trusted DRAM has been deprecated. Shared data is now always located at the base of Trusted SRAM.
  - BL2 Translation tables have been updated to map only the region of DRAM which is accessible to normal world. This is the region of the 2GB DDR-DRAM memory at 0x80000000 excluding the top 16MB. The top 16MB is accessible to only the secure world.
  - BL3-2 can now reside in the top 16MB of DRAM which is accessible only to the secure world. This can be done by setting the build flag `FVP_TSP_RAM_LOCATION` to the value `dram`.
- Separate translation tables are created for each boot loader image. The `IMAGE_BLx` build options are used to do this. This allows each stage to create mappings only for areas in the memory map that it needs.
- A Secure Payload Dispatcher (OPTEED) for the OP-TEE Trusted OS has been added. Details of using it with TF-A can be found in *OP-TEE Dispatcher*

10.11.2 Issues resolved since last release

- The Juno port has been aligned with the FVP port as follows.
  - Support for reclaiming all BL1 RW memory and BL2 memory by overlaying the BL3-1/BL3-2 NOBITS sections on top of them has been added to the Juno port.
  - The top 16MB of the 2GB DDR-DRAM memory at 0x80000000 is configured using the TZC-400 controller to be accessible only to the secure world.
  - The Arm GIC driver is used to configure the GIC-400 instead of using a GIC driver private to the Juno port.
  - PSCI `CPU_SUSPEND` calls that target a standby state are now supported.
  - The TZC-400 driver is used to configure the controller instead of direct accesses to the registers.
- The Linux kernel version referred to in the user guide has DVFS and HMP support enabled.
- DS-5 v5.19 did not detect Version 5.8 of the Cortex-A57-A53 Base FVPs in CADI server mode. This issue is not seen with DS-5 v5.20 and Version 6.2 of the Cortex-A57-A53 Base FVPs.

10.11.3 Known issues

- The Trusted Board Boot implementation is a prototype. There are issues with the modularity and scalability of the design. Support for a Trusted Watchdog, firmware update mechanism, recovery images and Trusted debug is absent. These issues will be addressed in future releases.
- The FVP and Juno ports do not use the hash of the ROTPK stored in the Trusted Key Storage registers to verify the ROTPK in the `plat_match_rotpk()` function. This prevents the correct establishment of the Chain of Trust at the first step in the Trusted Board Boot process.
- The version of the AEMv8 Base FVP used in this release resets the model instead of terminating its execution in response to a shutdown request using the PSCI `SYSTEM_OFF` API. This issue will be fixed in a future version of the model.

- GICv3 support is experimental. There are known issues with GICv3 initialization in the TF-A.
- While this version greatly reduces the on-chip RAM requirements, there are further RAM usage enhancements that could be made.
- The firmware design documentation for the Test Secure-EL1 Payload (TSP) and its dispatcher (TSPD) is incomplete. Similarly for the PSCI section.
- The Juno-specific firmware design documentation is incomplete.

10.12 Version 1.0

10.12.1 New features

- It is now possible to map higher physical addresses using non-flat virtual to physical address mappings in the MMU setup.
- Wider use is now made of the per-CPU data cache in BL3-1 to store:
  - Pointers to the non-secure and secure security state contexts.
  - A pointer to the CPU-specific operations.
  - A pointer to PSCI specific information (for example the current power state).
  - A crash reporting buffer.
- The following RAM usage improvements result in a BL3-1 RAM usage reduction from 96KB to 56KB (for FVP with TSPD), and a total RAM usage reduction across all images from 208KB to 88KB, compared to the previous release.
  - Removed the separate `early_exception` vectors from BL3-1 (2KB code size saving).
  - Removed NSRAM from the FVP memory map, allowing the removal of one (4KB) translation table.
  - Eliminated the internal `psci_suspend_context` array, saving 2KB.
  - Correctly dimensioned the PSCI `aff_map_node` array, saving 1.5KB in the FVP port.
  - Removed calling CPU mpidr from the bakery lock API, saving 160 bytes.
  - Removed current CPU mpidr from PSCI common code, saving 160 bytes.
  - Inlined the mmio accessor functions, saving 360 bytes.
  - Fully reclaimed all BL1 RW memory and BL2 memory on the FVP port by overlaying the BL3-1/BL3-2 NOBITS sections on top of these at runtime.
  - Made storing the FP register context optional, saving 0.5KB per context (8KB on the FVP port, with TSPD enabled and running on 8 CPUs).
  - Implemented a leaner `tf_printf()` function, allowing the stack to be greatly reduced.
  - Removed coherent stacks from the codebase. Stacks allocated in normal memory are now used before and after the MMU is enabled. This saves 768 bytes per CPU in BL3-1.
  - Reworked the crash reporting in BL3-1 to use less stack.
  - Optimized the EL3 register state stored in the `cpu_context` structure so that registers that do not change during normal execution are re-initialized each time during cold/warm boot, rather than restored from memory. This saves about 1.2KB.

  - As a result of some of the above, reduced the runtime stack size in all BL images. For BL3-1, this saves 1KB per CPU.
- PSCI SMC handler improvements to correctly handle calls from secure states and from AArch32.
- CPU contexts are now initialized from the `entry_point_info`. BL3-1 fully determines the exception level to use for the non-trusted firmware (BL3-3) based on the SPSR value provided by the BL2 platform code (or otherwise provided to BL3-1). This allows platform code to directly run non-trusted firmware payloads at either EL2 or EL1 without requiring an EL2 stub or OS loader.
- Code refactoring improvements:
  - Refactored `fvp_config` into a common platform header.
  - Refactored the fvp gic code to be a generic driver that no longer has an explicit dependency on platform code.
  - Refactored the CCI-400 driver to not have dependency on platform code.
  - Simplified the IO driver so it's no longer necessary to call `io_init()` and moved all the IO storage framework code to one place.
  - Simplified the interface the the TZC-400 driver.
  - Clarified the platform porting interface to the TSP.
  - Reworked the TSPD setup code to support the alternate BL3-2 initialization flow where BL3-1 generic code hands control to BL3-2, rather than expecting the TSPD to hand control directly to BL3-2.
  - Considerable rework to PSCI generic code to support CPU specific operations.
- Improved console log output, by:
  - Adding the concept of debug log levels.
  - Rationalizing the existing debug messages and adding new ones.
  - Printing out the version of each BL stage at runtime.
  - Adding support for printing console output from assembler code, including when a crash occurs before the C runtime is initialized.
- Moved up to the latest versions of the FVPs, toolchain, EDK2, kernel, Linaro file system and DS-5.
- On the FVP port, made the use of the Trusted DRAM region optional at build time (off by default). Normal platforms will not have such a “ready-to-use” DRAM area so it is not a good example to use it.
- Added support for PSCI `SYSTEM_OFF` and `SYSTEM_RESET` APIs.
- Added support for CPU specific reset sequences, power down sequences and register dumping during crash reporting. The CPU specific reset sequences include support for errata workarounds.
- Merged the Juno port into the master branch. Added support for CPU hotplug and CPU idle. Updated the user guide to describe how to build and run on the Juno platform.

10.12.2 Issues resolved since last release

- Removed the concept of top/bottom image loading. The image loader now automatically detects the position of the image inside the current memory layout and updates the layout to minimize fragmentation. This resolves the image loader limitations of previously releases. There are currently no plans to support dynamic image loading.
- CPU idle now works on the publicized version of the Foundation FVP.
- All known issues relating to the compiler version used have now been resolved. This TF-A version uses Linaro toolchain 14.07 (based on GCC 4.9).

10.12.3 Known issues

- GICv3 support is experimental. The Linux kernel patches to support this are not widely available. There are known issues with GICv3 initialization in the TF-A.
- While this version greatly reduces the on-chip RAM requirements, there are further RAM usage enhancements that could be made.
- The firmware design documentation for the Test Secure-EL1 Payload (TSP) and its dispatcher (TSPD) is incomplete. Similarly for the PSCI section.
- The Juno-specific firmware design documentation is incomplete.
- Some recent enhancements to the FVP port have not yet been translated into the Juno port. These will be tracked via the tf-issues project.
- The Linux kernel version referred to in the user guide has DVFS and HMP support disabled due to some known instabilities at the time of this release. A future kernel version will re-enable these features.
- DS-5 v5.19 does not detect Version 5.8 of the Cortex-A57-A53 Base FVPs in CADI server mode. This is because the `<SimName>` reported by the FVP in this version has changed. For example, for the Cortex-A57x4-A53x4 Base FVP, the `<SimName>` reported by the FVP is `FVP_Base_Cortex_A57x4_A53x4`, while DS-5 expects it to be `FVP_Base_A57x4_A53x4`.

  The temporary fix to this problem is to change the name of the FVP in `sw/debugger/configdb/Boards/ARM FVP/Base_A57x4_A53x4/cadi_config.xml`. Change the following line:

  ```
  <SimName>System Generator:FVP_Base_A57x4_A53x4</SimName>
  ```

  to System Generator:FVP_Base_Cortex-A57x4_A53x4

  A similar change can be made to the other Cortex-A57-A53 Base FVP variants.

10.13 Version 0.4

10.13.1 New features

- Makefile improvements:
  - Improved dependency checking when building.
  - Removed `dump` target (build now always produces dump files).
  - Enabled platform ports to optionally make use of parts of the Trusted Firmware (e.g. BL3-1 only), rather than being forced to use all parts. Also made the `fip` target optional.
  - Specified the full path to source files and removed use of the `vpath` keyword.

- Provided translation table library code for potential re-use by platforms other than the FVPs.
- Moved architectural timer setup to platform-specific code.
- Added standby state support to PSCI cpu_suspend implementation.
- SRAM usage improvements:
  - Started using the `-ffunction-sections`, `-fdata-sections` and `--gc-sections` compiler/linker options to remove unused code and data from the images. Previously, all common functions were being built into all binary images, whether or not they were actually used.
  - Placed all assembler functions in their own section to allow more unused functions to be removed from images.
  - Updated BL1 and BL2 to use a single coherent stack each, rather than one per CPU.
  - Changed variables that were unnecessarily declared and initialized as non-const (i.e. in the .data section) so they are either uninitialized (zero init) or const.
- Moved the Test Secure-EL1 Payload (BL3-2) to execute in Trusted SRAM by default. The option for it to run in Trusted DRAM remains.
- Implemented a TrustZone Address Space Controller (TZC-400) driver. A default configuration is provided for the Base FVPs. This means the model parameter `-C bp.secure_memory=1` is now supported.
- Started saving the PSCI cpu_suspend 'power_state' parameter prior to suspending a CPU. This allows platforms that implement multiple power-down states at the same affinity level to identify a specific state.
- Refactored the entire codebase to reduce the amount of nesting in header files and to make the use of system/user includes more consistent. Also split platform.h to separate out the platform porting declarations from the required platform porting definitions and the definitions/declarations specific to the platform port.
- Optimized the data cache clean/invalidate operations.
- Improved the BL3-1 unhandled exception handling and reporting. Unhandled exceptions now result in a dump of registers to the console.
- Major rework to the handover interface between BL stages, in particular the interface to BL3-1. The interface now conforms to a specification and is more future proof.
- Added support for optionally making the BL3-1 entrypoint a reset handler (instead of BL1). This allows platforms with an alternative image loading architecture to re-use BL3-1 with fewer modifications to generic code.
- Reserved some DDR DRAM for secure use on FVP platforms to avoid future compatibility problems with non-secure software.
- Added support for secure interrupts targeting the Secure-EL1 Payload (SP) (using GICv2 routing only). Demonstrated this working by adding an interrupt target and supporting test code to the TSP. Also demonstrated non-secure interrupt handling during TSP processing.

10.13.2 Issues resolved since last release

- Now support use of the model parameter `-C bp.secure_memory=1` in the Base FVPs (see **New features**).
- Support for secure world interrupt handling now available (see **New features**).
- Made enough SRAM savings (see **New features**) to enable the Test Secure-EL1 Payload (BL3-2) to execute in Trusted SRAM by default.
- The tested filesystem used for this release (Linaro AArch64 OpenEmbedded 14.04) now correctly reports progress in the console.

- Improved the Makefile structure to make it easier to separate out parts of the TF-A for re-use in platform ports. Also, improved target dependency checking.

10.13.3 Known issues

- GICv3 support is experimental. The Linux kernel patches to support this are not widely available. There are known issues with GICv3 initialization in the TF-A.
- Dynamic image loading is not available yet. The current image loader implementation (used to load BL2 and all subsequent images) has some limitations. Changing BL2 or BL3-1 load addresses in certain ways can lead to loading errors, even if the images should theoretically fit in memory.
- TF-A still uses too much on-chip Trusted SRAM. A number of RAM usage enhancements have been identified to rectify this situation.
- CPU idle does not work on the advertised version of the Foundation FVP. Some FVP fixes are required that are not available externally at the time of writing. This can be worked around by disabling CPU idle in the Linux kernel.
- Various bugs in TF-A, UEFI and the Linux kernel have been observed when using Linaro toolchain versions later than 13.11. Although most of these have been fixed, some remain at the time of writing. These mainly seem to relate to a subtle change in the way the compiler converts between 64-bit and 32-bit values (e.g. during casting operations), which reveals previously hidden bugs in client code.
- The firmware design documentation for the Test Secure-EL1 Payload (TSP) and its dispatcher (TSPD) is incomplete. Similarly for the PSCI section.

10.14 Version 0.3

10.14.1 New features

- Support for Foundation FVP Version 2.0 added. The documented UEFI configuration disables some devices that are unavailable in the Foundation FVP, including MMC and CLCD. The resultant UEFI binary can be used on the AEMv8 and Cortex-A57-A53 Base FVPs, as well as the Foundation FVP.

  Note: The software will not work on Version 1.0 of the Foundation FVP.

- Enabled third party contributions. Added a new contributing.md containing instructions for how to contribute and updated copyright text in all files to acknowledge contributors.
- The PSCI CPU_SUSPEND API has been stabilised to the extent where it can be used for entry into power down states with the following restrictions:
  - Entry into standby states is not supported.
  - The API is only supported on the AEMv8 and Cortex-A57-A53 Base FVPs.
- The PSCI AFFINITY_INFO api has undergone limited testing on the Base FVPs to allow experimental use.
- Required C library and runtime header files are now included locally in TF-A instead of depending on the toolchain standard include paths. The local implementation has been cleaned up and reduced in scope.
- Added I/O abstraction framework, primarily to allow generic code to load images in a platform-independent way. The existing image loading code has been reworked to use the new framework. Semi-hosting and NOR flash I/O drivers are provided.

- Introduced Firmware Image Package (FIP) handling code and tools. A FIP combines multiple firmware images with a Table of Contents (ToC) into a single binary image. The new FIP driver is another type of I/O driver. The Makefile builds a FIP by default and the FVP platform code expect to load a FIP from NOR flash, although some support for image loading using semi- hosting is retained.

  Note: Building a FIP by default is a non-backwards-compatible change.

  Note: Generic BL2 code now loads a BL3-3 (non-trusted firmware) image into DRAM instead of expecting this to be pre-loaded at known location. This is also a non-backwards-compatible change.

  Note: Some non-trusted firmware (e.g. UEFI) will need to be rebuilt so that it knows the new location to execute from and no longer needs to copy particular code modules to DRAM itself.

- Reworked BL2 to BL3-1 handover interface. A new composite structure (bl31_args) holds the superset of information that needs to be passed from BL2 to BL3-1, including information on how handover execution control to BL3-2 (if present) and BL3-3 (non-trusted firmware).
- Added library support for CPU context management, allowing the saving and restoring of
  - Shared system registers between Secure-EL1 and EL1.
  - VFP registers.
  - Essential EL3 system registers.
- Added a framework for implementing EL3 runtime services. Reworked the PSCI implementation to be one such runtime service.
- Reworked the exception handling logic, making use of both SP_EL0 and SP_EL3 stack pointers for determining the type of exception, managing general purpose and system register context on exception entry/exit, and handling SMCs. SMCs are directed to the correct EL3 runtime service.
- Added support for a Test Secure-EL1 Payload (TSP) and a corresponding Dispatcher (TSPD), which is loaded as an EL3 runtime service. The TSPD implements Secure Monitor functionality such as world switching and EL1 context management, and is responsible for communication with the TSP.

  Note: The TSPD does not yet contain support for secure world interrupts.

  Note: The TSP/TSPD is not built by default.

10.14.2 Issues resolved since last release

- Support has been added for switching context between secure and normal worlds in EL3.
- PSCI API calls `AFFINITY_INFO` & `PSCI_VERSION` have now been tested (to a limited extent).
- The TF-A build artifacts are now placed in the `./build` directory and sub-directories instead of being placed in the root of the project.
- TF-A is now free from build warnings. Build warnings are now treated as errors.
- TF-A now provides C library support locally within the project to maintain compatibility between toolchains/systems.
- The PSCI locking code has been reworked so it no longer takes locks in an incorrect sequence.
- The RAM-disk method of loading a Linux file-system has been confirmed to work with the TF-A and Linux kernel version (based on version 3.13) used in this release, for both Foundation and Base FVPs.

10.14.3 Known issues

The following is a list of issues which are expected to be fixed in the future releases of TF-A.

- The TrustZone Address Space Controller (TZC-400) is not being programmed yet. Use of model parameter `-C bp.secure_memory=1` is not supported.
- No support yet for secure world interrupt handling.
- GICv3 support is experimental. The Linux kernel patches to support this are not widely available. There are known issues with GICv3 initialization in TF-A.
- Dynamic image loading is not available yet. The current image loader implementation (used to load BL2 and all subsequent images) has some limitations. Changing BL2 or BL3-1 load addresses in certain ways can lead to loading errors, even if the images should theoretically fit in memory.
- TF-A uses too much on-chip Trusted SRAM. Currently the Test Secure-EL1 Payload (BL3-2) executes in Trusted DRAM since there is not enough SRAM. A number of RAM usage enhancements have been identified to rectify this situation.
- CPU idle does not work on the advertised version of the Foundation FVP. Some FVP fixes are required that are not available externally at the time of writing.
- Various bugs in TF-A, UEFI and the Linux kernel have been observed when using Linaro toolchain versions later than 13.11. Although most of these have been fixed, some remain at the time of writing. These mainly seem to relate to a subtle change in the way the compiler converts between 64-bit and 32-bit values (e.g. during casting operations), which reveals previously hidden bugs in client code.
- The tested filesystem used for this release (Linaro AArch64 OpenEmbedded 14.01) does not report progress correctly in the console. It only seems to produce error output, not standard output. It otherwise appears to function correctly. Other filesystem versions on the same software stack do not exhibit the problem.
- The Makefile structure doesn't make it easy to separate out parts of the TF-A for re-use in platform ports, for example if only BL3-1 is required in a platform port. Also, dependency checking in the Makefile is flawed.
- The firmware design documentation for the Test Secure-EL1 Payload (TSP) and its dispatcher (TSPD) is incomplete. Similarly for the PSCI section.

10.15 Version 0.2

10.15.1 New features

- First source release.
- Code for the PSCI suspend feature is supplied, although this is not enabled by default since there are known issues (see below).

10.15.2 Issues resolved since last release

- The “psci” nodes in the FDTs provided in this release now fully comply with the recommendations made in the PSCI specification.

10.15.3 Known issues

The following is a list of issues which are expected to be fixed in the future releases of TF-A.

- The TrustZone Address Space Controller (TZC-400) is not being programmed yet. Use of model parameter `-C bp.secure_memory=1` is not supported.
- No support yet for secure world interrupt handling or for switching context between secure and normal worlds in EL3.
- GICv3 support is experimental. The Linux kernel patches to support this are not widely available. There are known issues with GICv3 initialization in TF-A.
- Dynamic image loading is not available yet. The current image loader implementation (used to load BL2 and all subsequent images) has some limitations. Changing BL2 or BL3-1 load addresses in certain ways can lead to loading errors, even if the images should theoretically fit in memory.
- Although support for PSCI `CPU_SUSPEND` is present, it is not yet stable and ready for use.
- PSCI API calls `AFFINITY_INFO` & `PSCI_VERSION` are implemented but have not been tested.
- The TF-A make files result in all build artifacts being placed in the root of the project. These should be placed in appropriate sub-directories.
- The compilation of TF-A is not free from compilation warnings. Some of these warnings have not been investigated yet so they could mask real bugs.
- TF-A currently uses toolchain/system include files like stdio.h. It should provide versions of these within the project to maintain compatibility between toolchains/systems.
- The PSCI code takes some locks in an incorrect sequence. This may cause problems with suspend and hotplug in certain conditions.
- The Linux kernel used in this release is based on version 3.12-rc4. Using this kernel with the TF-A fails to start the file-system as a RAM-disk. It fails to execute user-space `init` from the RAM-disk. As an alternative, the VirtioBlock mechanism can be used to provide a file-system to the kernel.

CHAPTER

ELEVEN

CHANGE LOG FOR UPCOMING RELEASE

This document contains a summary of the new features, changes, fixes and known issues to be included in the upcoming release of Trusted Firmware-A. The contents of this file will be moved to the collective change-log.rst file at the time of release code freeze.

11.1 Upcoming Release Version 2.4

Trusted Firmware-A Contributors, Please log all relevant new features, changes, fixes, and known issues for the upcoming release. For the CPU support, drivers, and tools sections please preface the log description with the relevant key word, example: "<CPU>: <CPU Support addition>". Use the RST format convention already used in the Change Log.

11.1.1 New Features

- **Arm Architecture**
  - Example: "Add support for Branch Target Identification (BTI)"
- **BL-specific**
  - Example: "Enhanced BL2 bootloader flow to load secure partitions based on firmware configuration data (fconf)."
- **Build System**
  - Example: "Modify FVP makefile for CPUs that support both AArch64/32"
- **CPU Support**
  - Example: "cortex-a55: Workaround for erratum 1221012"
- **Drivers**
  - Example: "console: Allow the console to register multiple times"
- **Libraries**
  - Example: "Introduce BTI support in Library at ROM (romlib)"
- **New Platforms Support**
  - Example: "qemu/qemu_sbsa: New platform support added for QEMU SBSA platform"
- **Platforms**
  - Example: "arm/common: Introduce wrapper functions to setup secure watchdog"

- **PSCI**
  - Example: “Adding new optional PSCI hook `pwr_domain_on_finish_late`”
- **Security**
  - Example: “UBSAN support and handlers”
- **Tools**
  - Example: “fiptool: Add support to build fiptool on Windows.”

11.1.2 Changed

- **Arm Architecture**
  - Example: “Refactor ARMv8.3 Pointer Authentication support code”
- **BL-Specific**
  - Example: “BL2: Invalidate dcache build option for BL2 entry at EL3”
- **Boot Flow**
  - Example: “Add helper to parse BL31 parameters (both versions)”
- **Drivers**
  - Example: “gicv3: Prevent pending G1S interrupt from becoming G0 interrupt”
- **Platforms**
  - Example: “arm/common: Shorten the Firmware Update (FWU) process”
- **PSCI**
  - Example: “PSCI: Lookup list of parent nodes to lock only once”
- **Secure Partition Manager (SPM)**
  - Example: “Move shim layer to TTBR1_EL1”
- **Security**
  - Example: “Refactor SPSR initialisation code”
- **Tools**
  - Example: “cert_create: Remove RSA PKCS#1 v1.5 support”

11.1.3 Resolved Issues

- **Arm Architecture**
  - Example: “Fix restoration of PAuth context”
- **BL-Specific**
  - Example: “Fix BL31 crash reporting on AArch64 only platforms”
- **Build System**
  - Example: “Remove several warnings reported with W=2 and W=1”
- **Code Quality**
  - Example: “Unify type of “cpu_idx” across PSCI module”

- **CPU Support**
  - Example: “cortex-a12: Fix MIDR mask”
- **Drivers**
  - Example: “scmi: Fix wrong payload length”
- **Library Code**
  - Example: “libc: Fix memchr implementation”
- **Platforms**
  - Example: “rpi: rpi3: Fix compilation error when stack protector is enabled”
- **Security**
  - Example: “AArch32: Disable Secure Cycle Counter”

11.1.4 Deprecations

- **Common Code**
  - Example: “Remove MULTI_CONSOLE_API flag and references to it”
- **Drivers**
  - Example: “console: Remove deprecated finish_console_register”
- **Secure Partition Manager (SPM):**
  - Example: “Prototype SPCI-based SPM (services/std_svc/spm) will be replaced with alternative methods of secure partitioning support.”

11.1.5 Known Issues

- **Build System**
  - dtb: DTB creation not supported when building on a Windows host.

    This step in the build process is skipped when running on a Windows host. A known issue from the 1.6 release.
- **Platforms**
  - arm/juno: System suspend from Linux does not function as documented in the user guide

    Following the instructions provided in the user guide document does not result in the platform entering system suspend state as expected. A message relating to the hdlcd driver failing to suspend will be emitted on the Linux terminal.
  - mediatek/mt6795: This platform does not build in this release

CHAPTER

TWELVE

GLOSSARY

This glossary provides definitions for terms and abbreviations used in the TF-A documentation.

You can find additional definitions in the Arm Glossary.

AArch32 32-bit execution state of the ARMv8 ISA

AArch64 64-bit execution state of the ARMv8 ISA

API Application Programming Interface

AT Address Translation

BTI Branch Target Identification. An Armv8.5 extension providing additional control flow integrity around indirect branches and their targets.

CoT

COT Chain of Trust

CSS Compute Sub-System

CVE Common Vulnerabilities and Exposures. A CVE document is commonly used to describe a publicly-known security vulnerability.

DS-5 Arm Development Studio 5

DSU DynamIQ Shared Unit

DT Device Tree

DTB Device Tree Blob

EHF Exception Handling Framework

EL Exception Level

FCONF Firmware Configuration Framework

FDT Flattened Device Tree

FFA Firmware Framework for A-class processors

FIP Firmware Image Package

FVP Fixed Virtual Platform

FWU FirmWare Update

GIC Generic Interrupt Controller

ISA Instruction Set Architecture

Linaro A collaborative engineering organization consolidating and optimizing open source software and tools for the Arm architecture.

MMU Memory Management Unit

MPAM Memory Partitioning And Monitoring. An optional Armv8.4 extension.

MPIDR Multiprocessor Affinity Register

MTE Memory Tagging Extension. An optional Armv8.5 extension that enables hardware-assisted memory tagging.

OEN Owning Entity Number

OP-TEE Open Portable Trusted Execution Environment. An example of a *TEE*

OTE Open-source Trusted Execution Environment

PAUTH Pointer Authentication. An optional extension introduced in Armv8.3.

PDD Platform Design Document

PMF Performance Measurement Framework

PSA Platform Security Architecture

PSCI Power State Coordination Interface

RAS Reliability, Availability, and Serviceability extensions. A mandatory extension for the Armv8.2 architecture and later. An optional extension to the base Armv8 architecture.

ROT Root of Trust

SCMI System Control and Management Interface

SCP System Control Processor

SDEI Software Delegated Exception Interface

SDS Shared Data Storage

SEA Synchronous External Abort

SiP

SIP Silicon Provider

SMC Secure Monitor Call

SMCCC *SMC* Calling Convention

SoC System on Chip

SP Secure Partition

SPD Secure Payload Dispatcher

SPM Secure Partition Manager

SSBS Speculative Store Bypass Safe. Introduced in Armv8.5, this configuration bit can be set by software to allow or prevent the hardware from performing speculative operations.

SVE Scalable Vector Extension

TBB Trusted Board Boot

TBBR Trusted Board Boot Requirements

TEE Trusted Execution Environment

TF-A Trusted Firmware-A

TF-M Trusted Firmware-M

TLB Translation Lookaside Buffer

TLK Trusted Little Kernel. A Trusted OS from NVIDIA.

TSP Test Secure Payload

TZC TrustZone Controller

UBSAN Undefined Behavior Sanitizer

UEFI Unified Extensible Firmware Interface

WDOG Watchdog

XLAT Translation (abbr.). For example, “XLAT table”.

CHAPTER

THIRTEEN

LICENSE

The software is provided under a BSD-3-Clause license (below). Contributions to this project are accepted under the same license with developer sign-off as described in the *Contributor's Guide*.

13.1 SPDX Identifiers

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/

13.2 Other Projects

This project contains code from other projects as listed below. The original license text is included in those source files.

- The libc source code is derived from FreeBSD and SCC. FreeBSD uses various BSD licenses, including BSD-3-Clause and BSD-2-Clause. The SCC code is used under the BSD-3-Clause license with the author's permission.
- The libfdt source code is disjunctively dual licensed (GPL-2.0+ OR BSD-2-Clause). It is used by this project under the terms of the BSD-2-Clause license. Any contributions to this code must be made under the terms of both licenses.
- The LLVM compiler-rt source code is disjunctively dual licensed (NCSA OR MIT). It is used by this project under the terms of the NCSA license (also known as the University of Illinois/NCSA Open Source License), which is a permissive license compatible with BSD-3-Clause. Any contributions to this code must be made under the terms of both licenses.
- The zlib source code is licensed under the Zlib license, which is a permissive license compatible with BSD-3-Clause.
- Some STMicroelectronics platform source code is disjunctively dual licensed (GPL-2.0+ OR BSD-3-Clause). It is used by this project under the terms of the BSD-3-Clause license. Any contributions to this code must be made under the terms of both licenses.

Trusted Firmware-A (TF-A) provides a reference implementation of secure world software for Armv7-A and Armv8-A, including a Secure Monitor executing at Exception Level 3 (EL3). It implements various Arm interface standards, such as:

- The Power State Coordination Interface (PSCI)
- Trusted Board Boot Requirements CLIENT (TBBR-CLIENT)
- SMC Calling Convention
- System Control and Management Interface (SCMI)
- Software Delegated Exception Interface (SDEI)

Where possible, the code is designed for reuse or porting to other Armv7-A and Armv8-A model and hardware platforms.

This release provides a suitable starting point for productization of secure world boot and runtime firmware, in either the AArch32 or AArch64 execution states.

Users are encouraged to do their own security validation, including penetration testing, on any secure world code derived from TF-A.

In collaboration with interested parties, we will continue to enhance *TF-A* with reference implementations of Arm standards to benefit developers working with Armv7-A and Armv8-A TrustZone technology.

CHAPTER

FOURTEEN

GETTING STARTED

The *TF-A* documentation contains guidance for obtaining and building the software for existing, supported platforms, as well as supporting information for porting the software to a new platform.

The **About** chapter gives a high-level overview of *TF-A* features as well as some information on the project and how it is organized.

Refer to the documents in the **Getting Started** chapter for information about the prerequisites and requirements for building *TF-A*.

The **Processes & Policies** chapter explains the project's release schedule and process, how security disclosures are handled, and the guidelines for contributing to the project (including the coding style).

The **Components** chapter holds documents that explain specific components that make up the *TF-A* software, the *Exception Handling Framework*, for example.

In the **System Design** chapter you will find documents that explain the design of portions of the software that involve more than one component, such as the *Trusted Board Boot* process.

Platform Ports provides a list of the supported hardware and software-model platforms that are supported upstream in *TF-A*. Most of these platforms also have additional documentation that has been provided by the maintainers of the platform.

The results of any performance evaluations are added to the **Performance & Testing** chapter.

Security Advisories holds a list of documents relating to *CVE* entries that have previously been raised against the software.

INDEX