ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

BSEC internal peripheral

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

Contents

1. BSEC internal peripheral 3
2. BSEC device tree configuration 9
3. Device tree 16
4. How to assign an internal peripheral to a runtime context 21
5. NVMEM overview 28
6. OP-TEE overview 36
7. STM32CubeMX 43
8. STM32MP15 resources 46
9. STM32MPU Embedded Software architecture overview 50
10. TF-A overview 54

Stable: 24.09.2019 - 14:06 / Revision: 24.09.2019 - 07:57

A quality version of this page, approved on *24 September 2019*, was based off this revision.

Contents

1 Article purpose 4
2 Peripheral overview 5
  2.1 Features 5
  2.2 Security support 5
3 Peripheral usage and associated software 6
  3.1 Boot time 6
  3.2 Runtime 6
    3.2.1 Overview 6
    3.2.2 Software frameworks 6
    3.2.3 Peripheral configuration 6
    3.2.4 Peripheral assignment 6
4 How to go further 8
5 References 9

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

The purpose of this article is to

- briefly introduce the BSEC peripheral and its main features
- indicate the level of security supported by this hardware block
- explain how each instance can be allocated to the three runtime contexts and linked to the corresponding software components
- explain, when necessary, how to configure the BSEC peripheral.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

2 Peripheral overview

The **BSEC** peripheral is used to control an OTP (one time programmable) fuse box, used for on-chip non-volatile storage for device configuration and security parameters.

2.1 Features

Refer to STM32MP15 reference manuals for the complete list of features, and to the software components, introduced below, to see which features are implemented.

2.2 Security support

The BSEC is a **secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

The BSEC is configured at boot time to set up platform security.

3.2 Runtime

3.2.1 Overview

The BSEC instance is a system peripheral and is controlled by the Arm® Cortex®-A7 secure:

Information

- BSEC lower OTP access can be made available to the Arm® Cortex®-A7 non-secure.
- Upper OTP access can be managed as exceptions (in Trusted Boot Chain only, using TF-A), via "secure monitor calls", managed by TF-A or by OP-TEE. Please refer to BSEC device tree configuration for more details.

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
Security	BSEC	OP-TEE BSEC driver	Linux NVMEM framework		

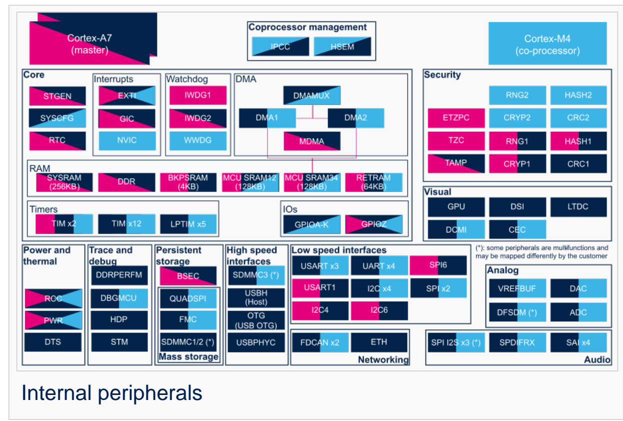
3.2.3 Peripheral configuration

The configuration is based on Device tree, please refer to BSEC device tree configuration article.
It can be applied by the firmware running in a secure context, done in TF-A or in OP-TEE.
It can also be configured by Linux® kernel, please refer to NVMEM overview article.

3.2.4 Peripheral assignment

Check boxes illustrate the possible peripheral allocations supported by STM32 MPU Embedded Software:

- means that the peripheral can be assigned () to the given runtime context.
- is used for system peripherals that cannot be unchecked because they are statically connected in the device.



Internal peripherals

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Refer to How to assign an internal peripheral to a runtime context for more information on how to assign peripherals manually or via STM32CubeMX.
The present chapter describes STMicroelectronics recommendations or choice of implementation. Additional possiblities might be described in STM32MP15 reference manuals.

Domain	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Security	BSEC	BSEC				

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

4 How to go further

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

5 References

Stable: 05.11.2020 - 15:18 / Revision: 04.11.2020 - 16:01

A quality version of this page, approved on *5 November 2020*, was based off this revision.

Contents

1 Article purpose 10
2 DT bindings documentation 11
3 DT configuration 12
3.1 DT configuration (STM32 level) 12
3.2 DT configuration (board level) 12
3.2.1 STM32MP1 BSEC node append 12
3.2.2 STM32MP1 BSEC node append (bootloader specific) 13
3.2.3 STM32MP1 driver node append 13
3.2.4 STM32MP1 nvmem_layout node (bootloader specific) 14
4 How to configure the DT using STM32CubeMX 15
5 References 16

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

Warning

This article explains how to configure BSEC at boot time.

This article describes the BSEC configuration performed using the device tree mechanism, which provides a hardware description of the BSEC peripheral.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 DT bindings documentation

Generic information about NVMEM is available in the NVMEM overview.

The following binding-related documentation explains how to write device tree files for BSEC:

- TF-A: *tf-a/docs/devicetree/bindings/soc/st,stm32-romem.txt"*[1]
- Linux® BSEC devicetree bindings: Documentation/devicetree/bindings/nvmem/st,stm32-romem.txt[2]
- Linux® generic NVMEM devicetree bindings: Documentation/devicetree/bindings/nvmem/nvmem.yaml[3] and Documentation /devicetree/bindings/nvmem/nvmem-consumer.yaml[4]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 DT configuration

This hardware description is a combination of the **STM32 microprocessor** device tree files (*.dtsi* extension) and **board** device tree files (*.dts* extension). See the Device tree for an explanation of the device-tree file split.

STM32CubeMX can be used to generate the board device tree. Refer to How to configure the DT using STM32CubeMX for more details.

3.1 DT configuration (STM32 level)

The STM32MP1 BSEC node is located in the file *stm32mp151.dtsi*[5] (see Device tree for further explanation).

```
/ {
...
        soc {
...
                bsec: nvmem@5c005000 {
                        compatible = "st,stm32mp15-bsec";
                        reg = <0x5c005000 0x400>;
                        #address-cells = <1>;
                        #size-cells = <1>;

                        part_number_otp: part_number_otp@4 {
                                reg = <0x4 0x1>;
                        };
                        ts_cal1: calib@5c {
                                reg = <0x5c 0x2>;
                        };
                        ts_cal2: calib@5e {
                                reg = <0x5e 0x2>;
                        };
                };
...
        };
...
};
```

Please refer to the NVMEM overview for the bindings common with the Linux® kernel.

3.2 DT configuration (board level)

3.2.1 STM32MP1 BSEC node append

The board definition in the device tree may include some additional board-specific OTP declarations:

```
&bsec {
        board_id: board_id@ec {
                reg = <0xec 0x4>;
                st,non-secure-otp;
        };
};
```

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

With only 32 lower NVMEM 32-bit data words, the software needs to manage exceptions in order to allow some upper OTPs to be accessed by the non-secure world, through secure world services for very specific needs. The user can add an OTP declaration in the device tree, using the "st,non-secure-otp" property, with a 32-bit length granularity (that is, 4 bytes).

3.2.2 STM32MP1 BSEC node append (bootloader specific)

The bootloader-specific STM32MP1 BSEC node append data is located in the file *stm32mp151.dtsi* [6] for TF-A (see Device tree for further explanation).
This completes NVMEM data providers, for bootloader-specific purposes only, either for a driver, or the platform itself.

```
bsec: nvmem@5c005000 {
        compatible = "st,stm32mp15-bsec";
        reg = <0x5c005000 0x400>;
        #address-cells = <1>;
        #size-cells = <1>;

        cfg0_otp: cfg0_otp@0 {
                reg = <0x0 0x1>;
        };
        part_number_otp: part_number_otp@4 {
                reg = <0x4 0x1>;
        };
        monotonic_otp: monotonic_otp@10 {
                reg = <0x10 0x4>;
        };
        nand_otp: nand_otp@24 {
                reg = <0x24 0x4>;
        };
        uid_otp: uid_otp@34 {
                reg = <0x34 0xc>;
        };
        package_otp: package_otp@40 {
                reg = <0x40 0x4>;
        };
        hw2_otp: hw2_otp@48 {
                reg = <0x48 0x4>;
        };
        ts_cal1: calib@5c {
                reg = <0x5c 0x2>;
        };
        ts_cal2: calib@5e {
                reg = <0x5e 0x2>;
        };
        pkh_otp: pkh_otp@60 {
                reg = <0x60 0x20>;
        };
        mac_addr: mac_addr@e4 {
                reg = <0xe4 0x8>;
                st,non-secure-otp;
        };
};
```

Please see the "st,non-secure-otp" definition in the previous section above. No more spare field declaration here.

3.2.3 STM32MP1 driver node append

The driver can directly consume NVMEM data cells, as described in NVMEM overview.
The CPU0 device is a good example, with a dedicated OTP containing part number information.
The device node is located in the *stm32mp151.dtsi*[5] file.

```
cpu0: cpu@0 {
        compatible = "arm,cortex-a7";
        device_type = "cpu";
        reg = <0>;
        clocks = <&scmi0_clk CK_SCMI0_MPU>;
        clock-names = "cpu";
        operating-points-v2 = <&cpu0_opp_table>;
        nvmem-cells = <&part_number_otp>;
        nvmem-cell-names = "part_number";
        #cooling-cells = <2>;
};
```

With these nvmem-cells / nvmem-cell-names properties, the CPU0 device can easily find the OTP number, in order to access part number information.

3.2.4 STM32MP1 nvmem_layout node (bootloader specific)

The STM32MP1 nvmem_layout node gathers all NVMEM platform-dependent layout information, including OTP names and phandles, in order to allow easy access for data consumers, using pre-defined string in the nvmem-cell-names property.

```
nvmem_layout: nvmem_layout@0 {
        compatible = "st,stm32mp1-nvmem-layout";
        nvmem-cells = <&cfg0_otp>,
                      <&part_number_otp>,
                      <&monotonic_otp>,
                      <&nand_otp>,
                      <&uid_otp>,
                      <&package_otp>,
                      <&hw2_otp>;

        nvmem-cell-names = "cfg0_otp",
                           "part_number_otp",
                           "monotonic_otp",
                           "uid_otp",
                           "nand_otp",
                           "package_otp",
                           "hw2_otp";
};
```

With this new node, the platform can easily find the OTP numbers, in order to access all the necessary information.

4 How to configure the DT using STM32CubeMX

The STM32CubeMX tool can be used to configure the STM32MPU device and get the corresponding platform configuration device tree files.
STM32CubeMX may not support all the properties described in the documents listed in DT bindings documentation above. If so, the tool inserts **user sections** in the generated device tree. These sections can then be edited to add some properties that are preserved from one generation to another. Refer to the STM32CubeMX user manual for further information.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

5 References

Please refer to the following links for additional information:

- docs/devicetree/bindings/soc/st,stm32-romem.txt TF-A BSEC binding information file
- Documentation/devicetree/bindings/nvmem/st,stm32-romem.txt
- Documentation/devicetree/bindings/nvmem/nvmem.yaml
- Documentation/devicetree/bindings/nvmem/nvmem-consumer.yaml
- 5.05.1 arch/arm/boot/dts/stm32mp151.dtsi : STM32MP151 Linux kernel device tree files
- fdts/stm32mp151.dtsi STM32MP151 TF-A device tree files

Stable: 05.11.2021 - 11:08 / Revision: 05.11.2021 - 11:05

A quality version of this page, approved on *5 November 2021*, was based off this revision.

Contents

1 Purpose 17
1.1 Device tree basis 17
1.2 Source files 17
1.3 Bindings 17
1.4 Build 18
1.5 Tools 18
2 STM32 19
3 How to go further 20
4 References 21

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Purpose

The objective of this chapter is to give general information about the device tree.

An extract of the **device tree specification**[1] explains it as follows:

"A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent. ... Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time."

In other words, a device tree describes the hardware that can not be located by probing.

1.1 Device tree basis

This webinar will give the foundations of device tree applied to STM32MP1 products and boards. This is highly recommended to start from this if you are beginner on this subject.

- Device Tree for STM32MP [2]

1.2 Source files

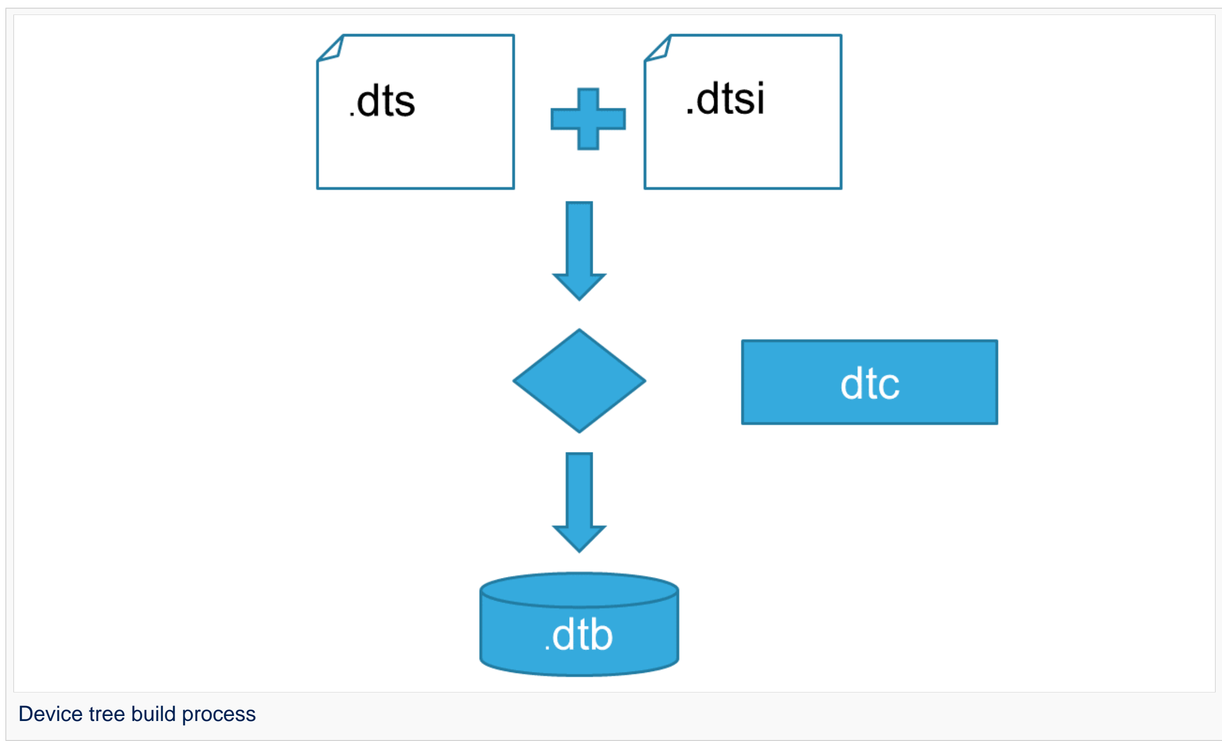
- **.dts:** The device tree source (DTS). This format is a textual representation of a device tree in a form that can be processed by DTC (Device Tree Compiler) into a binary file expected by software components: Linux® Kernel, U-Boot and TF-A.
- **.dtsi:** Source files that can be included from a DTS file.
- **.h:** Header files that can be included from DTS and DTSI files.

1.3 Bindings

The device tree data structures and properties are named **bindings**. Those bindings are described in:

- The Device tree specification[1] for generic bindings.
- The software component documentations:
  - Linux® Kernel: Linux kernel device tree bindings
  - U-Boot: doc/device-tree-bindings/
  - TF-A: TF-A device tree bindings

1.4 Build



Device tree build process

- A tool named DTC[3](Device Tree Compiler) allows compiling the DTS sources into a binary.
  - input file: the *.dts* file described in section above (that includes itself one or several .dtsi and .h files).
  - output file: the *.dtb* file described in section above.

DTC source code is located here[4]. DTC tool is also available directly in particular software components: **Linux Kernel, U-Boot, TF-A ....** For those components, the device tree building is directly integrated in the component build process.

If dts files use some defines, dts files should be preprocessed before being compiled by DTC.

1.5 Tools

The device tree compiler offers also some tools:

- **fdtdump:** Print a readable version of a flattened device tree file (dtb)
- **fdtget:** Read properties from a device tree
- **fdtput:** Write properties to a device tree
- ...

There are several ways to get those tools:

- In the device tree compiler project source code[4]
- Directly in software components: **Kernel, u-boot, tf-a ...**
- Available in Debian package[5]

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 STM32

For STM32MP1, the device tree is used by three software components: Linux® kernel, U-Boot and TF-A.

The device tree is part of the OpenSTLinux distribution. It can also be generated by STM32CubeMX tool.

To have more information about the device tree usage on STM32MP1 (how the device tree source files are split, how to find the device tree source files per software components, how is STM32CubeMX generating the device tree ...) see STM32MP15 device tree page.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 How to go further

- Device Tree Reference[6] - eLinux.org
- Device Tree usage[7] - eLinux.org

4 References

- [1.01.1] https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.2(latest) ,Device tree specification
- https://www.youtube.com/watch?v=a9CZ1Uk3OYQ, Device Tree for STM32MP
- https://git.kernel.org/pub/scm/utils/dtc/dtc.git/tree/Documentation/manual.txt(master), DTC manual
- [4.04.1] https://git.kernel.org/pub/scm/utils/dtc/dtc.git(master) ,DTC source code
- https://packages.debian.org/search?keywords=device-tree-compiler(master) ,DTC debian package
- Device Tree Reference, eLinux.org
- Device Tree Usage, eLinux.org

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Stable: 08.03.2021 - 16:13 / Revision: 16.02.2021 - 17:11

A quality version of this page, approved on *8 March 2021*, was based off this revision.

Contents

1 Article purpose 22
2 Introduction 23
3 STM32CubeMX generated assignment 24
4 Manual assignment 26
4.1 TF-A 26
4.2 U-boot 26
4.3 Linux kernel 27
4.4 STM32Cube 27
4.5 OP-TEE 28

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Article purpose

This article explains how to configure the software that assigns a peripheral to a runtime context.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 Introduction

A peripheral can be **assigned** to a runtime context via the configuration defined in the device tree. The device tree can be either generated by the STM32CubeMX tool or edited manually.

On STM32MP15 line devices, the assignement can be strengthened by a hardware mechanism: the ETZPC internal peripheral, which is configured by the TF-A boot loader. The ETZPC internal peripheral isolates the peripherals for the Cortex-A7 secure or the Cortex-M4 context. The peripherals assigned to the Cortex-A7 non-secure context are visible from any context, without any isolation.

The components running on the platform after TF-A execution (such as U-Boot, Linux, STM32Cube and OP-TEE) must have a **configuration** that is consistent with the assignment and the isolation configurations.

The following sections describe how to configure TF-A, U-Boot, Linux and STM32Cube accordingly.

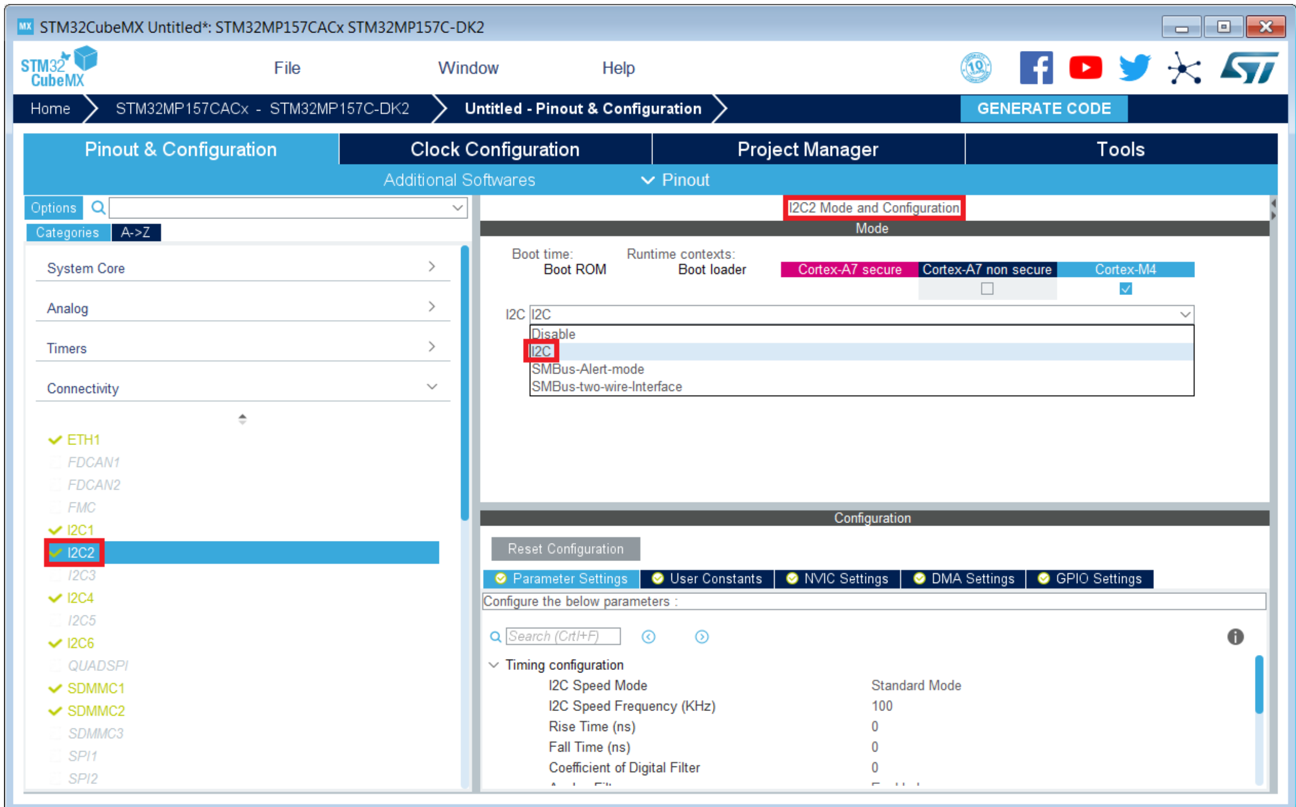
Information

Beyond the peripherals assignment, explained in this article, it is also important to understand How to configure system resources (i.e clocks, regulator, gpio,...), shared between the Cortex-A7 and Cortex-M4 contexts

3 STM32CubeMX generated assignment

The screenshot below shows the STM32CubeMX user interface:

- **I2C2** peripheral is selected, on the left
- **I2C2 Mode and Configuration** panel, on the right, shows that this I2C instance can be assigned to the Cortex-A7 non-secure or the Cortex-M4 (that is selected) runtime context
- **I2C** mode is enabled in the drop down menu



The context assignment table is displayed inside each peripheral **Mode and Configuration** panel but it is possible to display it for all the peripherals in the **Options** menu via the **Show contexts** option

The **GENERATE CODE** button, on the top right, produces the following:

- The **TF-A device tree** with the ETZPC configuration that isolates the I2C2 instance (in the example) for the Cortex-M4 context. This same device tree can be used by **OP-TEE**, when enabled
- The **U-Boot device tree** widely inherited from the Linux one, just below
- The **Linux kernel device tree** with the I2C node disabled for Linux and enabled for the coprocessor
- The **STM32Cube project** with I2C2 HAL initialization code

The Manual assignment section, just below, illustrates what STM32CubeMX is generating as it follows the same example.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

In addition of this generation, the user may have to manually complete the system resources configuration in the user sections embedded in the STM32CubeMX generated device tree. Refer to How to configure system resources for details.

4 Manual assignment

This section gives step by step instructions, per software components, to manually perform the peripherals assignments.
It takes the same I2C2 example as the previous section, that showed how to use STM32CubeMX, in order to make the move from one approach to the other easier.

Information

The assignments combinations described in the STM32MP15 peripherals overview article are naturally supported by STM32MPU Embedded Software distribution. Note that the STM32MP15 reference manual may describe more options that would require embedded software adaptations

4.1 TF-A

The assignment follows the ETZPC device tree configuration, with below possible values:

- **DECPROT_S_RW** for the Cortex-A7 secure (Secure OS like OP-TEE)
- **DECPROT_NS_RW** for the Cortex-A7 non-secure (Linux)
  - As stated earlier in this article, there is no hardware isolation for the Cortex-A7 non-secure so this value allows accesses from any context
- **DECPROT_MCU_ISOLATION** for the Cortex-M4 (STM32Cube)

Example:

```
@etzpc: etzpc@5C007000 {
       st,decprot = <
           DECPROT(STM32MP1_ETZPC_I2C2_ID, DECPROT_MCU_ISOLATION, DECPROT_UNLOCK)
       >;
};
```

Information

The value **DECPROT_NS_RW** can be used with **DECPROT_LOCK** as last parameter. In Cortex-M4 context, this specific configuration allows the generation of an error in the resource manager utility while trying to use on Cortex-M4 side a peripheral that is assigned to the Cortex-A7 non-secure context. If **DECPROT_UNLOCK** is used, then the utility allows the Cortex-M4 to use a peripheral that is assigned to the Cortex-A7 non-secure context.

4.2 U-boot

No specific configuration is needed in U-Boot to configure the access to the peripheral.

Information

U-Boot does not perform any check with regards to ETZPC configuration before accessing to a peripheral. In case of inconsistency an illegal access is generated.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Information

U-Boot checks the consistency between ETZPC isolation configuration and Linux kernel device tree configuration to guarantee that Linux kernel do not access an unauthorized device. In order to avoid the access to an unauthorized device, the U-boot fixes up the Linux kernel device tree to disable the peripheral nodes which are not assigned to the Cortex-A7 non-secure context.

4.3 Linux kernel

Each assignable peripheral is declared twice in the Linux kernel device tree:

- Once in the **soc** node from arch/arm/boot/dts/stm32mp151.dtsi , corresponding to Linux assigned peripherals
  - Example: i2c2
- Once in the **m4_rproc** node from arch/arm/boot/dts/stm32mp15-m4-srm.dtsi , corresponding to the Cortex-M4 context.

Those nodes are disabled, by default.
  - Example: m4_i2c2

In the board device tree file (*.dts), each assignable peripheral has to be enabled only for the context to which it is assigned, in line with TF-A configuration.
As a consequence, a peripheral assigned to the Cortex-A7 secure has both nodes disabled in the Linux device tree.

Example:

```
&i2c2 {
        status = "disabled";
};
...
&m4_i2c2 {
        status = "okay";
};
```

Information

In addition of this assignment, the user may have to complete the system resources configuration in the device tree nodes. Refer to How to configure system resources for details.

4.4 STM32Cube

There is no configuration to do on STM32Cube side regarding the assignment and isolation. Nevertheless, the resource manager utility, relying on ETZPC configuration, can be used to check that the corresponding peripheral is well assigned to the Cortex-M4 before using it.

Example:

```
int main(void)
{
...
 /* Initialize I2C2------------------------------------------------------ */
 /* Ask the resource manager for the I2C2 resource */
 ResMgr_Init(NULL, NULL);
 if (ResMgr_Request(RESMGR_ID_I2C2, RESMGR_FLAGS_ACCESS_NORMAL | \
```

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

```
                                                RESMGR_FLAGS_CPU1, 0, NULL) != RESMGR_OK)
  {
    Error_Handler();
  }
 ...
  if (HAL_I2C_Init(&I2C2) != HAL_OK)
  {
    Error_Handler();
  }
 }
```

4.5 OP-TEE

The OP-TEE OS may use STM32MP1 resources. OP-TEE STM32MP1 drivers register the device driver they intend to used in a secure context. This information is used to consolidate system configuration including secure hardening of configurable peripherals.

In most case, the OP-TEE driver probe relies on OP-TEE device tree porperty *secure-status = "okay"*.

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)

Stable: 04.11.2020 - 14:14 / Revision: 04.11.2020 - 14:04

A quality version of this page, approved on *4 November 2020*, was based off this revision.

This article introduces how NVMEM Linux® framework manages BSEC OTP data and how to read/write from/to it.

Contents

1 Framework purpose 29
2 System overview 30
2.1 Component description 30
2.2 API description 31
3 Configuration 32
3.1 Kernel configuration 32
3.2 Device tree configuration 32
4 How to use the framework 33
4.1 How to use NVMEM with sysfs interface 33
4.1.1 How to list NVMEM devices 33
4.1.2 How to read BSEC lower OTPs using NVMEM 33
4.1.3 How to read BSEC upper OTPs using NVMEM 33
4.1.4 How to write BSEC OTPs using NVMEM 34
5 How to trace and debug the framework 35
5.1 How to trace 35
6 References 36

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 Framework purpose

The NVMEM Linux® framework provides a generic interface for the device **non-volatile memory data** such as:

- OTP (one-time programmable) fuses
- EEPROM

It offers kernel space and user space interfaces to read and/or write data such as analog calibration data or MAC address.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 System overview

3rd Party/Customer
ST
Community
Hardware
Direct link
Interface
Legend

NVMEM overview

NVMEM user (user space)
User terminal
Application
User space
sysfs
Kernel space
NVMEM user (kernel space)
Foo NVMEM user
NVMEM user API
NVMEM framework
NVMEM Core
NVMEM drivers
BSEC
Hardware
STM32
BSEC (OTP)
life.augmented

NVMEM system overview

2.1 Component description

- **NVMEM user** (user space)

The user can use the NVMEM sysfs interface, from a user terminal or a custom application, to read/write data from/to NVMEM device(s) from user space.

- **NVMEM user** (kernel space)

User drivers can use the NVMEM API to read/write data from/to NVMEM device(s) from kernel space (such as the analog calibration data used by an ADC driver).

- **NVMEM framework** (kernel space)

The NVMEM core provides sysfs interface and NVMEM API. They can be used to implement NVMEM user and NVMEM controller drivers.

- **NVMEM drivers** (kernel space)

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Provider drivers such as BSEC Linux® driver that exposes OTP data to the core.

- **NVMEM hardware**

NVMEM controller(s) such as the *BSEC internal peripheral*[1]

2.2 API description

The NVMEM kernel documentation[2] describes:

- Kernel space API for NVMEM **providers** and NVMEM **consumers**.
- Userspace binary interface (sysfs).

See also *sysfs-bus-nvmem*[3] ABI documentation.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 Configuration

3.1 Kernel configuration

Activate NVMEM framework in the kernel configuration through the Linux® menuconfig tool, Menuconfig or how to configure kernel (CONFIG_NVMEM=y):

```
Device Drivers  --->
   [*] NVMEM Support  --->
      <*>   STMicroelectronics STM32 factory-programmed memory support
```

3.2 Device tree configuration

The NVMEM data device tree bindings describe:

- The location of non-volatile memory data
- The NVMEM data providers[4]
- The NVMEM data consumers[5]

The *BSEC internal peripheral*[1] device tree bindings are explained in BSEC device tree configuration article.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

4 How to use the framework

4.1 How to use NVMEM with sysfs interface

4.1.1 How to list NVMEM devices

The available NVMEM devices can be listed in sysfs:

```
# Example to list nvmem devices
Board $> ls /sys/bus/nvmem/devices/
stm32-romem0
```

The data content of an NVMEM device can be dumped to a binary file, and then displayed.

4.1.2 How to read BSEC lower OTPs using NVMEM

The **32 lower OTPs** can be read from non-secure when using either:

- the trusted boot chain (using TF-A)
- the basic boot chain (using U-Boot SPL)

```
# Example to read lower nvmem data content
Board $> dd if=/sys/bus/nvmem/devices/stm32-romem0/nvmem of=/tmp/file bs=4 count=32
# Example to display nvmem data content
Board $> hexdump -C -v /tmp/file
```

4.1.3 How to read BSEC upper OTPs using NVMEM

Only the 32 lower OTPs can be accessed when using the basic boot chain, as it doesn't implement secure services (CONFIG_HAVE_ARM_SMCCC). So this section concerns only the trusted boot chain (using TF-A) as SMC feature is available.

Default behavior for upper OTPS is normally restricted to security. If user needs more than the 32 lower OTPs, there is an exception management explained in BSEC device tree configuration.

It is then possible to access to some upper NVMEM information.

```
# Example to read the MAC address from upper OTP area, using secure services:
Board $> dd if=/sys/bus/nvmem/devices/stm32-romem0/nvmem of=/tmp/file skip=57 bs=4
count=2 status=none
Board $> hexdump -C -v /tmp/file
```

A dedicated chapter of the reference manual describes the OTP mapping.

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

4.1.4 How to write BSEC OTPs using NVMEM

Warning

The below examples show how to write data to an NVMEM device. This may cause unrecoverable damage to the STM32 device (for example when writing to an OTP area)

Information

Note that lower OTPs are using 2:1 redundancy, so they can be written bit per bit, whereas upper OTPs only support one time 32-bit programming.

Whatever the boot chain, the full lower NVMEM data content can be written as follows (if we suppose it has been previously read as described above, and updated directly in /tmp/file):

```
# Example to write lower nvmem data content
Board $> dd if=/tmp/file of=/sys/bus/nvmem/devices/stm32-romem0/nvmem bs=4 count=32
```

Only on Trusted boot chain, and under the condition the device tree authorizes it, an upper NVMEM data can be written. Example of 32-bit data word writing (filling it with ones) in OTP n°95:

```
# Create a 4 bytes length file filled with ones, e.g. 0xffffffff)
# Then, write it (32-bits, e.g. 4bytes) to OTP data 95
Board $> dd if=/dev/zero count=1 bs=4 | tr '\000' '\377' > file
Board $> dd if=file bs=4 seek=95 of=/sys/bus/nvmem/devices/stm32-romem0/nvmem
```

Information

When a new OTP value has been written using this SYSFS interface, it may be necessary to reboot the board before reading it back. The OTP value can't be read directly after a write because the OTP value is read in a shadow area not directly in the OTP area.

5 How to trace and debug the framework

5.1 How to trace

Ftrace can be used to trace the NVMEM framework:

```
Board $> cd /sys/kernel/debug/tracing
Board $> cat available_filter_functions | grep nvmem             # Show available filter 
functions
rtc_nvmem_register
rtc_nvmem_unregister
nvmem_reg_read
bin_attr_nvmem_read
...
```

Enable the kernel function tracer, then start using nvmem and display the result:

```
Board $> echo function > current_tracer
Board $> echo "*nvmem*" > set_ftrace_filter                      # Trace all nvmem filter 
functions
Board $> echo 1 > tracing_on                                     # start ftrace
Board $> hexdump -C -v /sys/bus/nvmem/devices/stm32-romem0/nvmem # dump nvmem
00000000  17 00 00 00 01 80 00 00  00 00 00 00 00 00 00 00  |................|
...
Board $> echo 0 > tracing_on                                     # stop ftrace
Board $> cat trace
# tracer: function
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
         hexdump-478   [000] ....   423.502278: bin_attr_nvmem_read <-sysfs_kf_bin_read
         hexdump-478   [000] ....   423.502290: nvmem_reg_read <-bin_attr_nvmem_read
         hexdump-478   [000] ....   423.515804: bin_attr_nvmem_read <-sysfs_kf_bin_read
```

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

6 References

- [1.01.1] BSEC internal peripheral
- Documentation/driver-api/nvmem.rst , NVMEM subsytem kernel documentation
- Documentation/ABI/stable/sysfs-bus-nvmem , NVMEM ABI documentation
- Documentation/devicetree/bindings/nvmem/nvmem.yaml , NVMEM device tree bindings
- Documentation/devicetree/bindings/nvmem/nvmem-consumer.yaml , NVMEM consumer device tree bindings

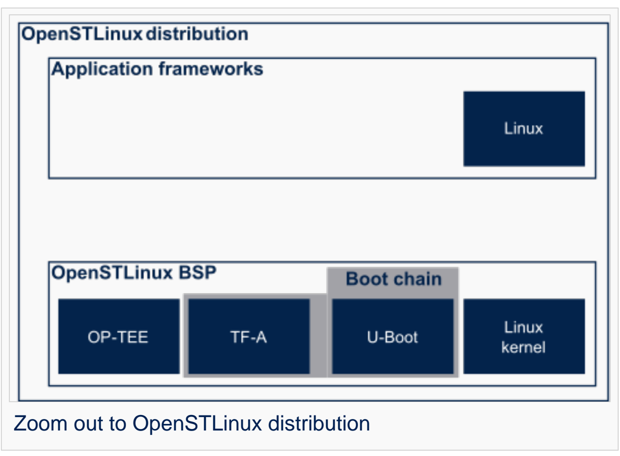
Stable: 13.05.2020 - 08:56 / Revision: 13.05.2020 - 08:54

A quality version of this page, approved on *13 May 2020*, was based off this revision.

Contents

1 Overview of the OP-TEE open source project 37
2 Architecture 38
  2.1 OP-TEE core 38
  2.2 OP-TEE trusted libraries 38
  2.3 TEE Linux driver 39
  2.4 TEE Client API 39
  2.5 TEE supplicant 39
  2.6 Host tools 39
3 Booting with OP-TEE 40
4 Invoking the OP-TEE services from Linux based OS 41
5 Experiencing OP-TEE on a target 42
6 References 43

1 Overview of the OP-TEE open source project



Zoom out to OpenSTLinux distribution

OP-TEE allows the development and integration of secure services and applications under trusted execution environments, that is execution environments isolated from the Linux® -based OS.

Description extracted from the OP-TEE site[1]:

> *"OP-TEE is an open source project, which contains a full implementation to make up a complete Trusted Execution Environment using the ARM® TrustZone®. technology. OP-TEE meets the GlobalPlatform TEE System Architecture specification. It also provides the TEE Internal core API v1.1 as defined by the GlobalPlatform TEE Standard for the development of Trusted Applications. OP-TEE Trusted OS is accessible from the Linux based OS using the GlobalPlatform TEE Client API Specification v1.0, which also is used to trigger secure execution of applications within the TEE."*

OP-TEE is delivered under a BSD style license and can run secure (trusted) applications without restriction on their licensing model.

The OP-TEE project is maintained by the Linaro Security Working Group.

- OP-TEE official site[1]
- OP-TEE source repositories [2][3][4]
- OP-TEE documentation[5]

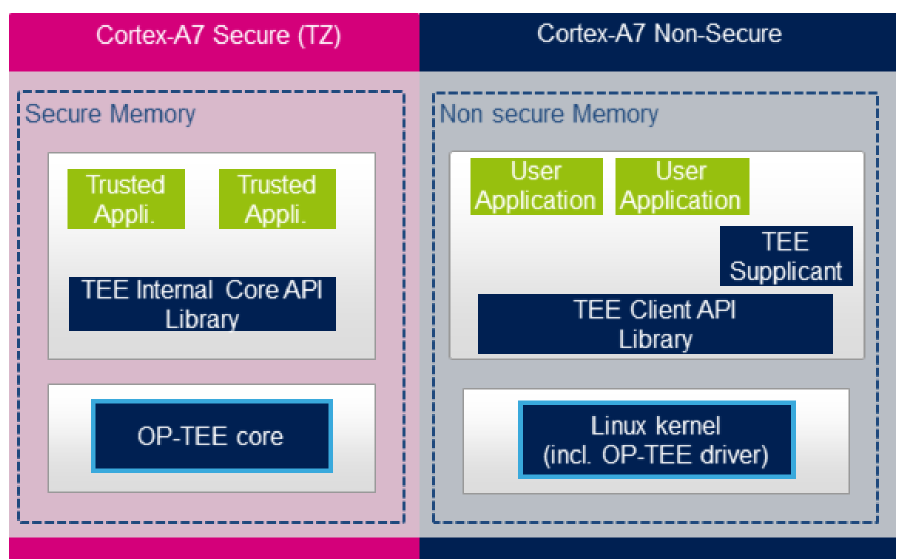
GlobalPlatform Device TEE specifications (TEE Client API, TEE Internal Core API and few more) is available from the GlobalPlatform site[6].

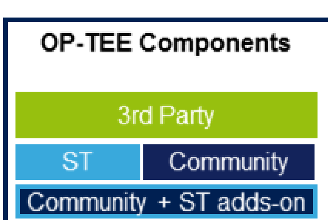
ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 Architecture

The OP-TEE project includes several secure and non-secure embedded components, as well as some tools for development and debugging purposes.

The figure below shows the main OP-TEE embedded components, namely the OP-TEE core and trusted application standard libraries on the secure side, and the Client API library, the OP-TEE supplicant daemon and the OP-TEE Linux kernel driver on the non-secure side.





2.1 OP-TEE core

The main OP-TEE component is the OP-TEE core. The OP-TEE core execution is done in Arm® Cortex®-A secure state while the non-secure world (likely a Linux based OS) is done in the non-secure state of the processor. The OP-TEE core executes in secure privileged (kernel) mode, while trusted applications are executed in secure user mode.

OP-TEE can load signed trusted applications stored in the Linux OS file system or embedded in the OP-TEE core boot image.

On devices with secure external memory, the OP-TEE core runs as a monolithic image in the secure memory. On devices with a small secure memory, the OP-TEE core can run in paging-on-demand configuration: a small resident agent is loaded in the small secure memory and can securely page-in/page-out data from/to the non-secure (or less secure) external memory.

OP-TEE core source files can be found from optee_os repository [2].

2.2 OP-TEE trusted libraries

OP-TEE embeds utility libraries for trusted application development including the GlobaPlatform Device TEE Internal Core API Library, which provides the standard services a trusted application can expect from the TEE. OP-TEE supports the loading of static and dynamic libraries in the TEE.

The OP-TEE standard trusted application libraries source files can be found in the optee_os repository[2].

2.3 TEE Linux driver

The OP-TEE Linux driver is part of the Linux kernel since release 4.12.

The OP-TEE Linux driver is enabled via the CONFIG_OPTEE configuration directive through the usual Linux kernel configuration means. The driver can be probed thanks to a device tree node.

2.4 TEE Client API

The OP-TEE project embeds an implementation of the GlobaPlatform Device TEE Client API specification for Linux based OS. This TEE Client API specification is partly implemented as a userland library and partly as a Linux kernel OP-TEE driver. The API allows userland clients to invoke trusted applications and the OP-TEE core services exported to non-secure world with a standard API.

The OP-TEE Client API library source files can be found in the optee_client repository[3].

2.5 TEE supplicant

The OP-TEE core can rely on non-secure remote services. OP-TEE embeds an implementation of a non-secure userland supplicant, that can be invoked by the OP-TEE core through the OP-TEE Linux kernel driver. An example of such service is the access to a non-volatile media device that is controlled in the non-secure world.

The OP-TEE supplicant source files can be found in the optee_client repository[3].

2.6 Host tools

The OP-TEE optee_os component, once built, generates a so-called Trusted Application Development Kit to ease the development and integration of trusted applications on a target system. The Trusted Application Development Kit includes the libraries, with their header files and makefile scripts, that allow the generation of signed trusted applications from their respective source files.

Optee_os package also provides a tool to analyse call stack backtraces in case of trusted application and/or OP-TEE core crash. Refer to script **symbolize.py** in optee_os source tree[2].

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

3 Booting with OP-TEE

The OP-TEE core is a secure firmware. It must be booted prior to the non-secure world on Arm Cortex-A core(s). The secure bootloader must therefore load the OP-TEE core images in memory and run its initialization prior to executing the first booted non-secure image.

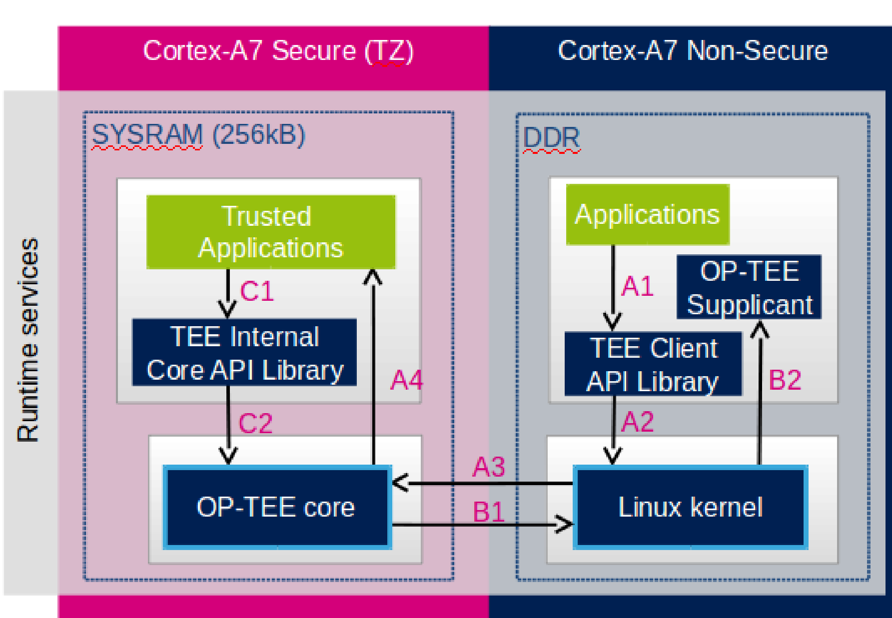
Refer to the target system boot sequences for more details.

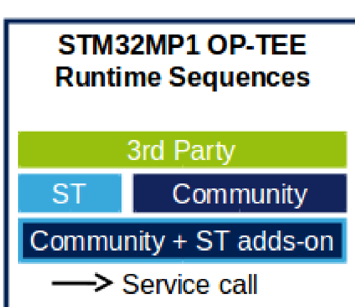
ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

4 Invoking the OP-TEE services from Linux based OS

Once the Linux kernel is booted, the OP-TEE core is already initialized and ready to serve.

The figure below shows the main run time sequences in which the OP-TEE can be involved.





Sequence A: an non-secure application invokes a service from a trusted application.

The non-secure application calls the TEE Client API library (**A1**), which in turns invokes (**A2**) the Linux kernel OP-TEE driver. The OP-TEE driver invokes the secure world (**A3**) and reaches the OP-TEE core. The last OP-TEE core transfers the request (**A4**) to the target trusted application. Once the trusted application has completed the request, the system branches back to the calling application with the request status.

If an invoked trusted application is not yet loaded into the TEE, the OP-TEE core loads it by calling remote services through the non-secure TEE supplicant as described in **sequence B** below.

In addition, any invocation of the TEE from the non-secure world must go through the Linux kernel OP-TEE driver.

Sequence B: the OP-TEE core must invoke a non-secure remote service.

The OP-TEE core invokes (**B1**) the Linux kernel OP-TEE driver which in turns notifies the TEE supplicant daemon (**B2**) for a request. Once the supplicant has completed the request, the system branches back to the OP-TEE core with the request status.

Sequence C: a trusted application invokes an OP-TEE core service.

Most of the services defined by the GlobaPlatform Device TEE Internal Core API must be executed in OP-TEE core privileged mode. The trusted application calls the corresponding service from the TEE Internal Core API library (**C1**), which issues a system call (**C2**) to the OP-TEE core. Once the core has completed the request, the system branches back to the calling trusted application with the request status.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

5 Experiencing OP-TEE on a target

First make sure your setup includes OP-TEE in the boot sequence. If the OP-TEE core console traces are enabled, you should see the OP-TEE banner after secure bootloader traces and before non-secure bootloader traces. The OP-TEE core banner looks like this:

```
I/TC:  OP-TEE version: <some-reference-version-info> #1 Mon Jun 25 08:59:21 UTC 2018 arm
I/TC:  Initialized
```

The Linux kernel boot traces also show the successful probing of the OP-TEE Linux kernel driver:

```
optee: probing for conduit method from DT.
optee: initialized driver
```

The OP-TEE non-secure components are stored in the file system:

- By default the TEE supplicant is installed at **/usr/bin/tee-supplicant**.
- By default, the TEE Client API library is installed at **/usr/lib/teec.so**.
- By default the TEE regression test tool is installed at **/usr/bin/xtest**.

In the default OP-TEE configuration, trusted applications are stored in the non-secure filesystem at **/lib/optee_armtz/*.ta**.

OP-TEE provides means to protect the trusted application binary images from corruption as image signature or installation in the OP-TEE secure storage. In any case, it is likely that Othe P-TEE core needs to invoke a non-secure service to retrieve the trusted application(s) from some non-secure filesystem data in order to load trusted application(s) in the TEE. This service requires the availability of the OP-TEE supplicant.

Therefore, once the non-secure OS has booted, it must launch the OP-TEE supplicant as a background daemon. Use the following shell command to start the OP-TEE supplicant from a booted Linux system, :

```
sh> tee-supplicant &
```

The OP-TEE package comes with some examples and regression tests. Use the following embedded shell command to run the regression tests:

```
sh> xtest
```

or to run only selective tests:

```
sh> xtest 1002     # Invokes some OP-TEE internal core services
sh> xtest 1004     # Invokes a trusted application loaded from the non-secure filesystem
```

6 References

- 1.01.1 https://op-tee.org
- 2.02.12.22.3 https://github.com/OP-TEE/optee_os
- 3.03.13.2 https://github.com/OP-TEE/optee_client
- https://github.com/OP-TEE/optee_test
- https://optee.readthedocs.io/
- https://globalplatform.org/

Stable: 23.09.2020 - 13:22 / Revision: 12.06.2020 - 13:25

A quality version of this page, approved on *23 September 2020*, was based off this revision.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

1 STM32CubeMX overview

This article describes STM32CubeMX, an official STMicroelectronics graphical software configuration tool.

The STM32CubeMX application helps developers to use the STM32 by means of a user interface, and guides the user through to the initial configuration of a firmware project.

It provides the means to:

- configure pin assignments, the clock tree, or internal peripherals
- simulate the power consumption of the resulting project
- configure and tune DDR parameters
- generate HAL initialization code for Cortex-M4
- generate the Device Tree for a Linux kernel, TF-A and U-Boot firmware for Cortex-A7

It uses a rich library of data from the STM32 microcontroller portfolio.

The application is intended to ease the initial development phase by helping developers to select the best product in terms of features and power.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 STM32CubeMX main features

- Peripheral and middleware parameters

  Presents options specific to each supported software component
- Peripheral assignment to processors

  Allows assignment of each peripheral to Cortex-A Secure, Cortex-A Non-Secure, or Cortex-M processors
- Power consumption calculator

  Uses a database of typical values to estimate power consumption, DMIPS, and battery life
- Code generation

  Makes code regeneration possible, while keeping user code intact
- Pinout configuration

  Enables peripherals to be chosen for use, and assigns GPIO and alternate functions to pins
- Clock tree initialization

  Chooses the oscillator and sets the PLL and clock dividers
- DDR tunning tool

  Ensures the configuration, testing, and tuning of the MPU DDR parameters. Using U-Boot-SPL Embedded Software.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 How to get STM32CubeMX

Please, refer to the following link STM32CubeMX to find STM32CubeMX, the Release Note, the User Manual and the product specification.

Das U-Boot -- the Universal Boot Loader (see U-Boot_overview)
Stable: 17.11.2021 - 16:41 / Revision: 17.11.2021 - 10:47

A quality version of this page, approved on *17 November 2021*, was based off this revision.

All the resources for the STM32MP1 Series are located in the Resources area of the STM32MP1 Series web page.

The resources below are referenced in some of the articles of this user guide.

Information

The different **STM32MP15** microprocessor **part numbers** available (with their corresponding internal peripherals, security options and packages) are described in the **STM32MP15 microprocessor part numbers**.

means that the document (or its version) is new compared to what was delivered within the previous ecosystem release.

Reference	Name	Link	Version
Application notes			
AN4803	High-speed SI simulations using IBIS and board-level simulations using HyperLynx® SI on STM32 MCUs and MPUs	AN4803.pdf	v2.0
AN5027	Interfacing PDM digital microphones using STM32 MCUs and MPUs	AN5027.pdf	v2.0
AN5031	Getting started with STM32MP15 Series hardware development	AN5031.pdf	v3.0
AN5036	Thermal management guidelines for STM32 applications	AN5036.pdf	v3.0
AN5109	STM32MP1 Series using low-power modes	AN5109.pdf	v4.0
AN5122	STM32MP1 Series DDR memory routing guidelines	AN5122.pdf	v3.0
AN5168	STM32MP1 series DDR configuration	AN5168.pdf	NEW v2.0
	USB Type-C™ Power Delivery using STM32xx Series MCUs and	AN522	

ST Restricted – Subject to Non-Disclosure Agreement – Do not copy

Referen ce	Name	Link	Version
Application notes			
AN5225	STM32xxx Series MPUs	5.pdf	NEW v5.0
AN5253	Migration of microcontroller applications from STM32F4x9 lines to STM32MP151, STM32MP153 and STM32MP157 lines microprocessor	AN525 3.pdf	v1.0
AN5256	STM32MP151, STM32MP153 and STM32MP157 discrete power supply hardware integration	AN525 6.pdf	v2.0
AN5260	STM32MP151/153/157 MPU lines and STPMIC1B integration on a battery powered application	AN526 0.pdf	v2.0
AN5275	USB DFU/USART protocols used in STM32MP1 Series bootloaders	AN527 5.pdf	v1.0
AN5284	STM32MP1 series system power consumption	AN528 4.pdf	v1.0
AN5348	FDCAN peripheral on STM32 devices	AN534 8.pdf	v1.0
AN5431	The STPMIC1 PCB layout guidlines	AN543 1.pdf	v1.0
AN5438	STM32MP1 Series lifetime estimates	AN543 8.pdf	v1.0
AN5510	Overview of the secure secret provisioning (SSP) on STM32MP1 Series	AN551 0.pdf	v1.0
Datasheets[1]			
DS1250 5	STM32MP157C/F datasheet (secure)	DS125 05.pdf	NEW v6.0
DS1250 4	STM32MP157A/D datasheet (basic)	DS125 04.pdf	NEW v6.0
DS1250 3	STM32MP153C/F datasheet (secure)	DS125 03.pdf	NEW v6.0
DS1250 2	STM32MP153A/D datasheet (basic)	DS125 02.pdf	NEW v6.0
DS1250 1	STM32MP151C/F datasheet (secure)	DS125 01.pdf	NEW v6.0
DS1250 0	STM32MP151A/D datasheet (basic)	DS125 00.pdf	NEW v6.0
DS1279 2	STPMIC1 datasheet	DS127 92.pdf	NEW v8.0

Reference	Name	Link	Version
Application notes			
Errata sheets			
ES0438	STM32MP15xx device errata	ES0438.pdf	v6.0
Reference manuals[1]			
RM0436	STM32MP157 reference manual (STM32MP157xxx advanced Arm®-based 32-bit MPUs)	RM0436.pdf	v5.0
RM0442	STM32MP153 reference manual (STM32MP153xxx advanced Arm®-based 32-bit MPUs)	RM0442.pdf	v5.0
RM0441	STM32MP151 reference manual (STM32MP151xxx advanced Arm®-based 32-bit MPUs)	RM0441.pdf	v5.0
Boards schematics			
MB1262 schematics	STM32MP157C-EV1 motherboard schematics MB1262-C01 board schematic (Evaluation board)	MB1262-C01.pdf	v1.0
MB1263 schematics	STM32MP157F-EV1 daughterboard schematics MB1263-C04 board schematic (Evaluation board)	MB1263-C04.pdf	v4.0
MB1230 schematics	DSI 720p LCD display daughterboard schematics MB1230-C board schematic (Evaluation board)	MB1230-C.pdf	v1.1
MB1379 schematics	Camera daughterboard schematics MB1379-A01 board schematic (Evaluation board)	MB1379-A01.pdf	v1.0
MB1272 schematics	STM32MP157x-DKx motherboard schematics MB1272-DK2-C01 board schematic (Discovery kit)	MB1272-C01.pdf	v1.0
MB1407 schematics	STM32MP157x-DKx daughterboard schematics MB1407-LCD-C01 board schematic (Discovery kit)	MB1407-C01.pdf	v1.0
Boards user manuals			
UM2535	STM32MP157x-EV1 evaluation board user manual	UM2535.pdf	v2.0
UM2534	STM32MP157x-DKx discovery board user manual	UM2534.pdf	v1.0
Tools user manuals			
UM2563	STM32CubeIDE installation guide	UM2563.pdf	NEW v3.0

Reference	Name	Link	Version
Application notes			
UM2579	Migration guide from System Workbench to STM32CubeIDE	UM2579.pdf	v1.0
UM2553	STM32CubeIDE quick start guide	UM2553.pdf	NEW v3.0
AN5360	Getting started with projects based on the STM32MP1 Series in STM32CubeIDE	AN5360.pdf	v1.0
UM2609	STM32CubeIDE user guide	UM2609.pdf	NEW v5.0
UM1718	STM32CubeMX user manual	UM1718.pdf	NEW v36.0
UM2237	STM32CubeProgrammer tool user manual	UM2237.pdf	NEW v17.0
UM2238	STM32 Trusted Package Creator tool user manual	UM2238.pdf	NEW v9.0
UM2542	STM32 Series Key Generator tool user manual	UM2542.pdf	NEW v2.0
UM2543	STM32 Series Signing tool user manual	UM2543.pdf	NEW v2.0

- 1.01.1 The part numbers are specified in STM32MP15 microprocessor part numbers

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

Archives

STM32MP15 release	ST documentation
STM32MP15-Ecosystem-v3.0.0	STM32MP15 resources - v3.0.0 page for the previous v3 ecosystem release
STM32MP15-Ecosystem-v2.1.0	STM32MP15 resources - v2.1.0 page for the v2 ecosystem releases (in archived wiki)
STM32MP15-Ecosystem-v2.0.0	STM32MP15 resources - v2.0.0 page for the v2 ecosystem releases (in archived wiki)
STM32MP15-Ecosystem-v1.2.0	STM32MP15 resources - v1.2.0 page for the v1 ecosystem releases (in archived wiki)
STM32MP15-Ecosystem-v1.1.0	STM32MP15 resources - v1.1.0 page for the v1 ecosystem releases (in archived wiki)
STM32MP15-Ecosystem-v1.0.0	STM32MP15 resources - v1.0.0 page for the v1 ecosystem releases (in archived wiki)

USB port or connector

Universal Synchronous/Asynchronous Receiver/Transmitter

Stable: 26.03.2021 - 11:32 / Revision: 12.03.2021 - 11:07

A quality version of this page, approved on *26 March 2021*, was based off this revision.

1 STM32MPU Embedded Software overview

The diagram below shows STM32MPU Embedded Software distribution main components:

- The **OpenSTLinux distribution**, running on the Arm® Cortex®-A, including:
  - The **OpenSTLinux BSP** with:
    - The boot chain based on TF-A and U-Boot.
    - The OP-TEE secure OS running on the Arm® Cortex®-A in secure mode.
    - The **Linux® kernel** running on the Arm® Cortex®-A in non-secure mode.
  - The **application frameworks** are composed of middlewares relying on the BSP and providing API, on **Linux** side, to run **Applications** that typically interact with the user via the display, the touchscreen, etc.
  - On **OP-TEE** side, the **Trusted Applications (TA)** relies on the OP-TEE core for secrets operations (not visible from the Linux and STM32Cube MPU Package)
- The **STM32Cube MPU Package** is running on the Arm® Cortex®-M: it is based on HAL drivers and middlewares, like other STM32 microcontrollers, completed with coprocessor management.

The figure below is clickable so that the user can directly jump to one of the sub-levels listed above.

STM32MPU Embedded Software architecture overview.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 Open Source Software (OSS) philosophy

The **Open source software** source code is released under a license in which the copyright holder grants users the rights to study, change and distribute the software to anyone and for any purpose[1].
STMicroelectronics maximizes the using of open source software and contributes to those communities. Notice that, due to the software review life cycle, it can take some time before getting all developments accepted in the communities, so STMicroelectronics can also temporarily provide some source code on github[2], until it is merged in the targeted repository.

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 References

- https://en.wikipedia.org/wiki/Open-source_software
- STM32MP1 Distribution Package

Stable: 22.04.2021 - 11:23 / Revision: 09.04.2021 - 13:17

A quality version of this page, approved on *22 April 2021*, was based off this revision.

Contents

1 Trusted Firmware-A 55
2 Architecture 56
3 Boot loader stages 58
3.1 BL1 58
3.2 BL2 58
3.2.1 FIP 58
3.2.2 Firmware Configuration 58
3.2.3 Authentication 59
3.3 BL32 59
4 References 60

1 Trusted Firmware-A

Trusted Firmware-A is a reference implementation of secure-world software provided by Arm®. It was first designed for Armv8-A platforms, and has been adapted to be used on Armv7-A platforms by STMicroelectronics. Trusted Firmware-A is part of the Trusted Firmware project that is an open governance community project hosted by Linaro.[1]

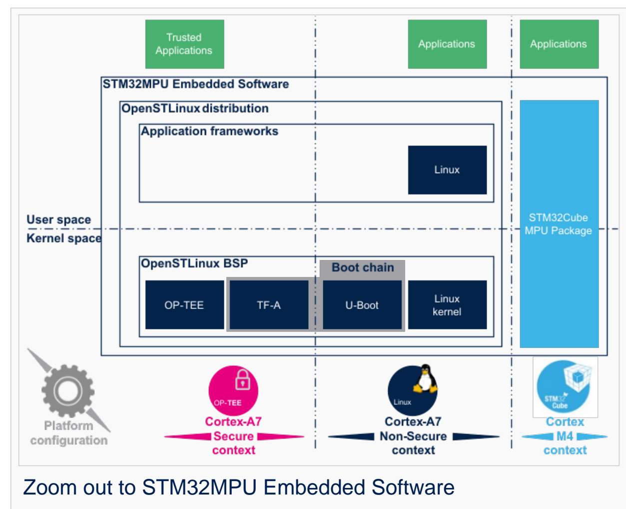
It is used as the first-stage boot loader (FSBL) on STM32 MPU platforms when using the trusted boot chain.

The code is open source, under a BSD-3-Clause license, and can be found on Linaro project page [2], including an up-to-date documentation about Trusted Firmware-A implementation [3].

Trusted Firmware-A also implements a set of features with various Arm interface standards:

- The power state coordination interface (PSCI) [4]
- SMC calling convention [5]
- System control and management interface [6]

Trusted Firmware-A is usually shortened to TF-A.



Zoom out to STM32MPU Embedded Software

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

2 Architecture

The global architecture of TF-A is explained in the Trusted Firmware-A design [7] document.

TF-A is divided into several binaries, each with a dedicated main role.
For 32-bit Arm processors (AArch32), the trusted boot is divided into four stages (in order of execution):

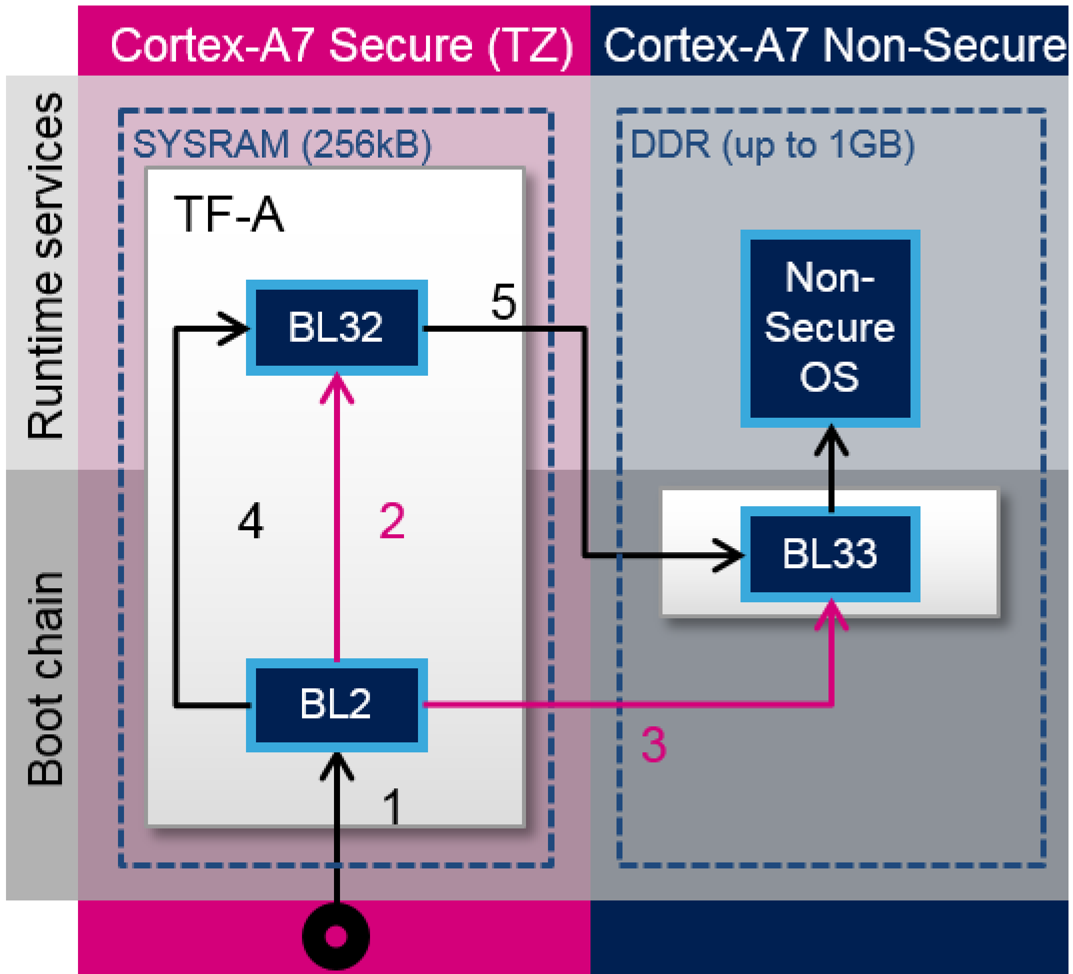
- Boot loader stage 1 (BL1) application processor trusted ROM
- Boot loader stage 2 (BL2) trusted boot firmware
- Boot loader stage 3-2 (BL32) runtime software
- Boot loader stage 3-3 (BL33) non-trusted firmware

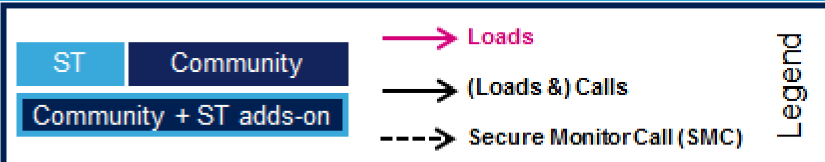
BL1, BL2 and BL32 are parts of TF-A.

Because STM32 MPU platforms uses a dedicated ROM code, the BL1 boot stage is then removed. ROM code expects the BL2 to run at EL3 execution level. This mode is selected when the BL2_AT_EL3 build flag is enabled.

BL33 is outside of TF-A. This is the first non-secure code loaded by TF-A. During the boot sequence, this is the secondary stage boot loader (SSBL). For STM32 MPU platforms, the SSBL is U-Boot by default.

TF-A can manage its configuration with a device tree. In the BL2 stage, it is a reduced version of the Linux kernel one, with only the required devices used during boot. It can be configured with STM32CubeMX.





TF-A loading steps:

1. ROM code loads and calls BL2
2. BL2 loads BL32
3. BL2 loads BL33
4. BL2 calls BL32
5. BL32 calls BL33

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

3 Boot loader stages

3.1 BL1

BL1 is the first stage executed, and is designed to act as ROM code; it is loaded and executed in internal RAM. It is not used for the STM32 MPU. As the STM32 MPU has its own proprietary ROM code, this part can be removed and BL2 is then the first TF-A binary to be executed.

3.2 BL2

BL2 is in charge of loading the next-stage images (secure and non secure). To achieve this role, BL2 has to initialize all the required peripherals.

- System components: clocks, DDR, ...
- Security components: Firewall
- Storage

BL2 offers different features to load and authenticate images.

At the end of its execution, after having loaded BL32 and the next boot stage (BL33), BL2 jumps to BL32.

3.2.1 FIP

The Firmware Image Package (FIP)[8] is a TF-A archive binary that encapsulates bootloader images into a single archive. It can also contain other data such as certificates that are required to complete the boot process. A dedicated driver drivers/io/io_fip.c able to read data from this package is part of the TF-A BL2.
FIP uses a specific layout based on a table of contents followed by payload data. It is synchronized between the driver and the host creation tool: Fiptool tools/fiptool/fiptool.c . This host tool is able to create a package, get info from the package, update, unpack or remove data in this package.

3.2.2 Firmware Configuration

The Firmware Configuration Framework (FCONF)[9] is a way to offer more flexibility in the firmware. It is used to provide most of the platform-specific data that were previously hard coded inside the firmware. This framework uses device tree (one or multiple) that are passed to the firmware during load processing. BL2 uses it to describe the chain of trust and the images list to be loaded.
Thanks to device tree usage, the configuration becomes dynamic at boot time. The current implementation uses the following device tree as framework entry:

- FW_CONFIG - The firmware configuration file. Hold properties shared across all BLx images. An example is the **dtb-registry** node, which contains the information about other binaries configuration (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).

3.2.3 Authentication

TF-A BL2 implements an authentication framework that uses a defined Chain of Trust (CoT) based on Arm TBBR[10] requirement to achieve a secure boot. The authentication is enabled as soon as the **TRUSTED_BOARD_BOOT** flag is defined. TF-A BL2 implements this CoT which is based on a Root of Trust Public Key (ROTPK). The CoT relies on a public key infrastructure generating self-signed certificate (following X509 v3 standard [11]). There is **no Certificate Authority (CA)** because the CoT is not established by verifying the validity of a certificate's issuer.

Different keys are used for this CoT:

- 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 certificates used in this CoT could be Key certificate or Content certificate.

- 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.

3.3 BL32

BL32 provides runtime secure services.

On Armv7 architecture, the BL32 must embed a Secure Monitor as it will be executed in the same privilege level (PL1-SVC Secure). TF-A provides a minimal monitor implementation: SP-MIN. It is described in the TF-A functionality list [3] as: "A minimal AArch32 Secure Payload (SP-MIN) to demonstrate PSCI [4] library integration with AArch32 EL3 Runtime Software."

This minimal implementation can be replaced with a trusted OS or trusted environment execution (TEE), such as OP-TEE that also embeds a secure monitor on Armv7. Both solutions (SP-MIN or OP-TEE) are supported by STMicroelectronics for STM32MP15.

BL32 acts as a secure monitor and thus provides secure services to non-secure OSs. These services are called by non-secure software with secure monitor calls [5].

This code is in charge of standard service calls, like PSCI [4] or SCMI [6].

It also provides STMicroelectronics proprietary services to access secure peripherals (with secure access control).

ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

4 References

- https://www.trustedfirmware.org/
- https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git
- 3.03.1 https://trustedfirmware-a.readthedocs.io/en/latest/
- 4.04.14.2 ARM Power State Coordination Interface
- 5.05.1 SMC Calling Convention (SMCCC)
- 6.06.1 Arm System Control and Management Interface
- https://trustedfirmware-a.readthedocs.io/en/latest/design/index.html
- https://trustedfirmware-a.readthedocs.io/en/latest/design/firmware-design.html?highlight=FIP#firmware-image-package-fip
- https://trustedfirmware-a.readthedocs.io/en/latest/components/fconf/index.html?highlight=FCONF
- https://trustedfirmware-a.readthedocs.io/en/latest/design/trusted-board-boot.html
- https://tools.ietf.org/rfc/rfc5280.txt