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

How to activate HSI and CSI oscillators calibration

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

Contents

1. How to activate HSI and CSI oscillators calibration 3
2. Clock device tree configuration - Bootloader specific 5
3. OP-TEE overview 19
4. RCC internal peripheral 26
5. STM32MP15 resources 32
6. TF-A overview 37
7. TIM internal peripheral 45

Stable: 03.10.2019 - 12:48 / Revision: 03.10.2019 - 12:47

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

A quality version of this page, approved on *3 October 2019*, was based off this revision.

Contents

1 Article purpose 4
2 How to activate the calibration 5
2.1 Configuring the timers 5
2.2 Enabling and configuring the calibration service 5

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

1 Article purpose

The purpose of this article is to explain how to calibrate the HSI and CSI oscillators in the RCC, by using the TIM internal peripheral **TIM12** and/or **TIM15** assigned to the secure monitor (TF-A or OP-TEE).

These clocks are internal oscillators whose frequency can be affected by temperature and voltage variations. To achieve a good clock accuracy, it is important to provide a mechanism to compensate the effects of these variations.

The clock calibration algorithm is based on the comparison of a timer (fed by HSI or CSI) and a clock is derived from the HSE clock that is considered as always accurate.

TIM12 input 1 is connected to hsi_cal_ck

TIM12 input 2 is connected to csi_cal_ck

TIM15 input 7 is connected to hsi_cal_ck

TIM15 input 8 is connected to csi_cal_ck

Refer to STM32MP15 reference manuals for detailed information on the timers.

The algorithm is implemented in the secure monitor. It compares both clocks and programs a correction factor in the RCC peripheral. There are various ways to trigger this service:

- periodically by the secure monitor itself
- upon kernel request through a dedicated SMC
- upon Arm®Cortex®-M4 request through a SEV

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

2 How to activate the calibration

This is done in the secure monitor device tree, in the stm32mp157c-<board>.dts file.

2.1 Configuring the timers

The timers used for calibration must be dedicated in the secure context. It cannot be used at the same time by the non-secure world.

Example: timer12-input1 is used for HSI and timer12-input2 for CSI

```
&timers12 {
            secure-status = "okay";
            st,hsi-cal-input = <1>;
            st,csi_cal-input = <2>;
};
```

```
&timers15 {
            secure-status = "disabled";
            st,hsi-cal-input = <7>;
            st,csi_cal-input = <8>;
};
```

2.2 Enabling and configuring the calibration service

This can be done by enabling options inside the clock device tree section

High Speed Internal oscillator (STM32 clock source) or High Speed Synchronous Serial Interface (MIPI® Alliance standard)

Multi Speed Internal oscillator (STM32 clock source)

High Speed External oscillator (STM32 clock source)

Reset and Clock Control

Secure Monitor Call

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. arm

Cortex®

Stable: 24.03.2021 - 07:55 / Revision: 16.02.2021 - 16:11

A quality version of this page, approved on *24 March 2021*, was based off this revision.

Contents

1 Article purpose 7
2 DT bindings documentation 8
3 DT configuration 9
3.1 DT configuration (STM32 level) 9

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

3.2 DT configuration (board level) 9
3.2.1 Clock node 9
3.2.1.1 Optional properties for "clk-lse" and "clk-hse" external oscillators 10
3.2.1.2 DT configuration for HSE 10
3.2.1.3 DT configuration for LSE 11
3.2.1.4 Optional property for "clk-hsi" internal oscillator 11
3.2.1.5 Clock node example 12
3.2.2 STM32MP1 clock node 13
3.2.2.1 Defining clock source distribution with st,clksrc property 13
3.2.2.2 Defining clock dividers with st,clkdiv property 14
3.2.2.3 Defining peripheral PLL frequencies with st,pll property 14
3.2.2.4 Defining peripheral kernel clock tree distribution with st,pkcs property 15
3.2.2.5 HSI and CSI clocks calibration 16
4 How to configure the DT using STM32CubeMX 17
5 References 18

1 Article purpose

This article describes the specific **RCC** internal peripheral configuration done by the first stage bootloader:

- TF-A for the Trusted boot chain
- U-Boot SPL DDR interactive mode for the DDR tuning tool

Regarding OP-TEE when it is embedded in the device, OP-TEE OS is booted by TF-A BL2, it is booted by TF-A BL2 bootstage. OP-TEE relies on TF-A BL2 bootstage for the RCC clock tree initial configuration. This article explicitily mentions OP-TEE when in information applies to OP-TEE secure world configuration.

This article explains how to configure the clock tree in the RCC at boot time.
You can then refer to the clock device tree configuration article to understand how to derive each internal peripheral clock tree in Linux®OS from the RCC clock tree.

The configuration is performed using the device tree mechanism that provides a hardware description of the RCC peripheral.

This clock tree is only used in the device tree of the boot chain FSBL; so in the TF-A device tree for OpenSTLinux official delivery (or in SPL only for the DDR tuning tool).

Even if the clock tree information is also present in the U-Boot device tree, it is not used during boot by this SSBL.

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

2 DT bindings documentation

The bootloader clock device tree bindings correspond to the vendor clock DT bindings used by the clk-stm32mp1 driver of the FSBL (TF-A or U-Boot SPL for DDR interactive mode), it is based on:

- binding described in Clock_device_tree_configuration
- bootloader specific properties described in #DT configuration

This binding document explains how to write the device tree files for clocks on the bootloader side:

- TF-A: *tf-a/docs/devicetree/bindings/clock/st,stm32mp1-rcc.txt"*[1]
- U-Boot SPL for DDR interactive mode: *doc/device-tree-bindings/clock/st,stm32mp1.txt*[2]

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 clock nodes are located in *stm32mp151.dtsi*[3] (see Device tree for more explanations):

- fixed-clock defined in clock node
- RCC node for #STM32MP1 clock node: clock generation and distribution.

```
 / {
 ...
         clocks {
                 clk_hse: clk-hse {
                         #clock-cells = <0>;
                         compatible = "fixed-clock";
                         clock-frequency = <24000000>;
                 };

 ...
           };
 ...
         soc {
 ...
                 rcc: rcc@50000000 {
                         compatible = "st,stm32mp1-rcc", "syscon";
                         reg = <0x50000000 0x1000>;
                         #clock-cells = <1>;
                         #reset-cells = <1>;
                         interrupts = <GIC_SPI 5 IRQ_TYPE_LEVEL_HIGH>;
                 };
...

         };

 };
```

Please refer to clock device tree configuration for the bindings common with Linux® kernel.

3.2 DT configuration (board level)

3.2.1 Clock node

Note: this section applies to OP-TEE that gets input clocks frequency value from the device tree description it boots upon.

The clock tree is also based on five fixed clocks in the clock node. They are used to define the state of associated STM32MP1 oscillators:

- clk-lsi
- clk-lse

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

- clk-hsi
- clk-hse
- clk-csi

Please refer to clock device tree configuration for detailed information.

At boot time, the clock tree initialization performs the following tasks:

- enabling of the oscillators present in the device tree and not disabled (node with status="disabled"),
- disabling of the HSI oscillator if the node is absent or disabled (HSI is always activated by the ROM code).

3.2.1.1 *Optional properties for "clk-lse" and "clk-hse" external oscillators*

For external oscillator HSE and LSE, the default clock configuration is an external crystal/ceramic resonator.

Four optional fields are supported:

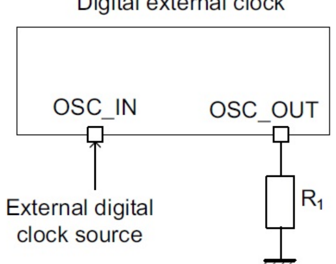
- "st,bypass" configures the external analog clock source (set HSEBYP, LSEBYP),
- "st,digbypass" configures the external digital clock source (set DIGBYP and HSEBYP, LSEBYP),
- "st,css" activates the clock security system (HSECSSON, LSECSSON),
- "st,drive" (LSE only) contains the value of the drive for the oscillator (see LSEDRV_ defined in the file *stm32mp1-clksrc.h*[4]).

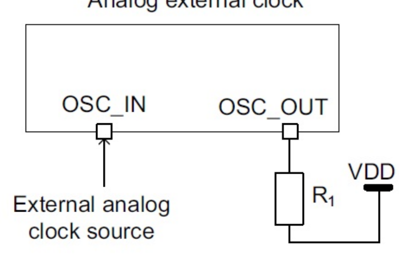
3.2.1.2 *DT configuration for HSE*

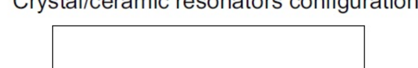
The HSE can accept an external crystal/ceramic or external clock source on OSC_IN, digital or analog : the user needs to select the correct frequency and the correct configuration in the device tree, corresponding to the hardware setup.

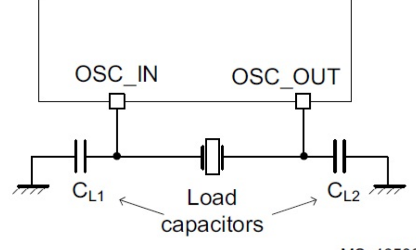
All the ST boards are using a digital external clock configuration (so device tree with = st,digbypass).

For example with the same 24MHz frequency, we have 3 configurations:









MSv46580V2

- Digital external clock = st,digbypass

```
/ {
        clocks {
                clk_hse: clk-hse {
                        #clock-cells = <0>;
                        compatible = "fixed-clock";
                        clock-frequency = <24000000>;
                        st,digbypass;
                };
};
```

- Analog external clock = st,bypass

```
/ {
        clocks {
                clk_hse: clk-hse {
                        #clock-cells = <0>;
```

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

```
                        compatible = "fixed-clock";
                        clock-frequency = <24000000>;
                        st,bypass;
                };
        };
```

- Crystal/ ceramic resonators configuration

```
        / {
                clocks {
                        clk_hse: clk-hse {
                                #clock-cells = <0>;
                                compatible = "fixed-clock";
                                clock-frequency = <24000000>;
                        };
        };
```

3.2.1.3 DT configuration for LSE

Below an example of LSE on board file with 32,768kHz crystal resonators, the drive set to medium high and with activated clock security system.

```
        / {
                clocks {
                        clk_lse: clk-lse {
                                #clock-cells = <0>;
                                compatible = "fixed-clock";
                                clock-frequency = <32768>;
                                st,css;
                                st,drive = <LSEDRV_MEDIUM_HIGH>;
                        };
        };
```

3.2.1.4 Optional property for "clk-hsi" internal oscillator

The HSI clock frequency is internally fixed to 64 MHz for the STM32MP15 devices.

In the device tree, clk-hsi is the clock after HSIDIV divider (more information on clk_hsi can be found in the RCC chapter in the reference manual).
As a result the frequency of this fixed clock is used to compute the expected HSIDIV for the clock tree initialization.

Below an example with HSIDIV = 1/1:

```
        / {
                clocks {
                        clk_hsi: clk-hsi {
                                #clock-cells = <0>;
                                compatible = "fixed-clock";
                                clock-frequency = <64000000>;
                        };
        };
```

Below an example with HSIDIV = 1/2:

```
	/ {
		clocks {
			clk_hsi: clk-hsi {
				#clock-cells = <0>;
				compatible = "fixed-clock";
				clock-frequency = <32000000>;
			};
	};
```

3.2.1.5 Clock node example

Note: this section applies to OP-TEE OS clock drivers.

An example of clocks node with:

- all oscillators switched on (HSE, HSI, LSE, LSI, CSI)
- HSI at 64MHZ (HSIDIV = 1/1)
- HSE using a digital external clock at 24MHz
- LSE using an external crystal at 32.768kHz (the typical frequency)

We highlight the customized parts:

```
/ {
	clocks {
		clk_hse: clk-hse {
			#clock-cells = <0>;
			compatible = "fixed-clock";
			clock-frequency = <24000000>;
			st,digbypass;
		};

		clk_hsi: clk-hsi {
			#clock-cells = <0>;
			compatible = "fixed-clock";
			clock-frequency = <64000000>;
		};

		clk_lse: clk-lse {
			#clock-cells = <0>;
			compatible = "fixed-clock";
			clock-frequency = <32768>;
		};

		clk_lsi: clk-lsi {
			#clock-cells = <0>;
			compatible = "fixed-clock";
			clock-frequency = <32000>;
		};

		clk_csi: clk-csi {
			#clock-cells = <0>;
			compatible = "fixed-clock";
			clock-frequency = <4000000>;
		};
	};
};
```

So the resulting board device tree, based on SoC device tree "stm32mp151.dtsi", is :

```
#include "stm32mp151.dtsi"
&clk_hse {
        clock-frequency = <24000000>;
        st,digbypass;
};

&clk_hsi {
        clock-frequency = <64000000>;
};

&clk_lse {
        clock-frequency = <32768>;
};
```

It is the configuration used by TF-A for STM32MP15 boards.

3.2.2 STM32MP1 clock node

Please refer to clock device tree configuration for information on how to specify the number of cells in a clock specifier.

The bootloader performs a global clock initialization, as described below. The information related to a given board can be found in the board specific device tree files listed in clock node.

The bootloader uses other properties for RCC node ("st,stm32mp1-rcc" compatible):

- secure-status: related to TZEN bit configuration in RCC security register that allows to restrict RCC and PWR registers write access
- st,clksrc: clock source configuration array
- st,clkdiv: clock divider configuration array
- st,pll: specific PLL configuration
- st,pkcs: peripheral kernel clock distribution configuration array.

All the available clocks are defined as preprocessor macros in *stm32mp1-clks.h*[5] and can be used in device tree sources.

Note: this section partially applies to OP-TEE OS clock drivers in that OP-TEE OS clock drivers consider only property *secure-status* over those listed above.

3.2.2.1 Defining clock source distribution with st,clksrc property

This property can be used to configure the clock distribution tree. When used, it must describe the whole distribution tree.

There are nine clock source selectors for the STM32MP15 devices. They must be configured in the following order: MPU, AXI, MCU, PLL12, PLL3, PLL4, RTC, MCO1, and MCO2.

The clock source configuration values are defined by the CLK_<NAME>_<SOURCE> macros located in *stm32mp1-clksrc.h*[4].

Example:

```
        st,clksrc = <
                CLK_MPU_PLL1P
                CLK_AXI_PLL2P
                CLK_MCU_PLL3P
                CLK_PLL12_HSE
                CLK_PLL3_HSE
                CLK_PLL4_HSE
                CLK_RTC_LSE
                CLK_MCO1_DISABLED
                CLK_MCO2_DISABLED
        >;
```

3.2.2.2 *Defining clock dividers with st,clkdiv property*

This property can be used to configure the value of the clock main dividers. When used, it must describe the whole clock divider tree.

There are 11 dividers values for the STM32MP15 devices. They must be configured in the following order: MPU, AXI, MCU, APB1, APB2, APB3, APB4, APB5, RTC, MCO1 and MCO2.

Each divider value uses the DIV coding defined in the RCC associated register, RCC_xxxDIVR. In most cases, this value is the following:

- 0x0: not divided
- 0x1: division by 2
- 0x2: division by 4
- 0x3: division by 8
- ...

Note that the coding differs for RTC MCO1 and MCO2:

- 0x0: not divided
- 0x1: division by 2
- 0x2: division by 3
- 0x3: division by 4
- ...

Example:

```
        st,clkdiv = <
                1 /*MPU*/
                0 /*AXI*/
                0 /*MCU*/
                1 /*APB1*/
                1 /*APB2*/
                1 /*APB3*/
                1 /*APB4*/
                2 /*APB5*/
                23 /*RTC*/
                0 /*MCO1*/
                0 /*MCO2*/
        >;
```

3.2.2.3 *Defining peripheral PLL frequencies with st,pll property*

This property can be used to configure PLL frequencies.

The PLL children nodes for PLL1 to PLL4 (see reference manual for details) are associated with an index from 0 to 3 (st,pll@0 to st,pll@3).

PLL2, PLL3 or PLL4 are off when their associated nodes are absent or deactivated.

The configuration of PLL1, the source clock of Cortex-A7 core, with st,pll@0 node, is optional as TF-A automatically selects the most suitable operating point for the platform (please refer to How to change the CPU frequency). The node st,pll@0 node should be absent; it is only used if you want to override the PLL1 properties computed by TF-A (clock spreading for example).

Below the available properties for each PLL node:

- cfg contains the PLL configuration parameters in the following order: DIVM, DIVN, DIVP, DIVQ, DIVR, output.

  DIVx values are defined as in RCC:

  - 0x0: bypass (division by 1)

- 0x1: division by 2
- 0x2: division by 3
- 0x3: division by 4
- ...

Output contains a bitfield for each output value (1:ON / 0:OFF)

- BIT(0) output P : DIVPEN
- BIT(1) output Q : DIVQEN
- BIT(2) output R : DIVREN

Note: PQR(p,q,r) macro can be used to build this value with p, q, r = 0 or 1.

- frac: fractional part of the multiplication factor (optional, when absent PLL is in integer mode).
- csg contains the clock spreading generator parameters (optional) in the following order: MOD_PER, INC_STEP and SSCG_MODE.

MOD_PER: modulation period adjustment

INC_STEP: modulation depth adjustment

SSCG_MODE: Spread spectrum clock generator mode, defined in *stm32mp1-clksrc.h*[4]:

- SSCG_MODE_CENTER_SPREAD = 0
- SSCG_MODE_DOWN_SPREAD = 1

Example:

```
	pll2: st,pll@1 {
		compatible = "st,stm32mp1-pll";
		reg = <1>;
		cfg = < 1 43 1 0 0 PQR(0,1,1) >;
		csg = < 10 20 1 >;
	};
	pll3: st,pll@2 {
		compatible = "st,stm32mp1-pll";
		reg = <2>;
		cfg = < 2 85 3 13 3 0 >;
		csg = < 10 20 SSCG_MODE_CENTER_SPREAD >;
	};
	pll4: st,pll@3 {
		compatible = "st,stm32mp1-pll";
		reg = <3>;
		cfg = < 2 78 4 7 9 3 >;
	};
```

3.2.2.4 Defining peripheral kernel clock tree distribution with st,pkcs property

This property can be used to configure the peripheral kernel clock selection.

It is a list of peripheral kernel clock source identifiers defined by the CLK_<KERNEL-CLOCK>_<PARENT-CLOCK> macros in the *stm32mp1-clksrc.h*[4] header file.

st,pkcs may not list all the kernel clocks. No specific order is required.

Example:

```
	st,pkcs = <
		CLK_STGEN_HSE
		CLK_CKPER_HSI
```

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

```
                CLK_USBPHY_PLL2P
                CLK_DSI_PLL2Q
                CLK_I2C46_HSI
                CLK_UART1_HSI
                CLK_UART24_HSI
        >;
```

3.2.2.5 HSI and CSI clocks calibration

Note: this section applies to OP-TEE OS clock calibration support.

The calibration is an optional feature that can be enabled from the device tree. It allows requesting the HSI or CSI clock calibration by several means:

- SiP SMC service
- Periodic calibration every X seconds
- Interrupt raised by the MCU

This feature requires that a hardware timer is assigned to the calibration sequence.

A dedicated interrupt must be defined using "mcu_sev" name to start a calibration on detection of an interrupt raised by the MCU.

- st,hsi-cal: used to enable HSI clock calibration feature.
- st,csi-cal; used to enable CSI clock calibration feature.
- st,cal-sec: used to enable periodic calibration at specified time intervals from the secure monitor. The time interval must be given in seconds. If not specified, a calibration is only processed for each incoming request.

Example:

```
        &rcc {
                st,hsi-cal;
                st,csi-cal;
                st,cal-sec = <15>;
                secure-interrupts = <GIC_SPI 144 IRQ_TYPE_LEVEL_HIGH>,
                                    <GIC_SPI 145 IRQ_TYPE_LEVEL_HIGH>;
                interrupt-names = "mcu_sev", "wakeup";
        };
```

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

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.

The STM32CubeMX may not support all the properties described in the above DT bindings documentation paragraph.
If so, the tool inserts **user sections** in the generated device tree.
These sections can then be edited to add some properties and they are preserved from one generation to another.

Refer to 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/clock/st,stm32mp1-rcc.txt TF-A clock binding information file
- doc/device-tree-bindings/clock/st,stm32mp1.txt U-Boot SPL for DDR interactive mode clock binding information file
- fdts/stm32mp151.dtsi (for TF-A), arch/arm/dts/stm32mp15-no-scmi.dtsi (for U-Boot SPL for DDR interactive mode): STM32MP151 device tree files
- 4.04.14.24.3 include/dt-bindings/clock/stm32mp1-clksrc.h (for TF-A), include/dt-bindings/clock/stm32mp1-clksrc.h (for U-Boot SPL for DDR interactive mode): STM32MP1 DT bindings clock source files
- include/dt-bindings/clock/stm32mp1-clks.h (for TF-A), include/dt-bindings/clock/stm32mp1-clks.h (for U-Boot SPL for DDR interactive mode): STM32MP1 DT bindings clock identifier files

Doubledata rate (memory domain)

Open Portable Trusted Execution Environment

Operating System

Trusted Firmware for Arm® Cortex®-A

Boot Loader stage 2

Reset and Clock Control

Linux® is a registered trademark of Linus Torvalds.

First Stage Boot Loader

Secondary Program Loader, *Also known as* ***U-Boot SPL***

Second Stage Boot Loader

Device Tree

Generic Interrupt Controller

Serial Peripheral Interface

High Speed Internal oscillator (STM32 clock source) or High Speed Synchronous Serial Interface (MIPI® Alliance standard)

Read Only Memory

High Speed External oscillator (STM32 clock source)

Low Speed External oscillator (STM32 clock source)

Low Speed Internal oscillator (STM32 clock source)

Multi Speed Internal oscillator (STM32 clock source)

Microprocessor Unit

Microcontroller Unit (MCUs have internal flash memory and are intended to operate with a minimum amount of external support ICs. They commonly are a self-contained, system-on-chip (SoC) designs.)

Real Time Clock

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

Cortex®

System Time Generator

Display Serial Interface (MIPI® Alliance standard)

Silicon Provider

Secure Monitor Call

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.

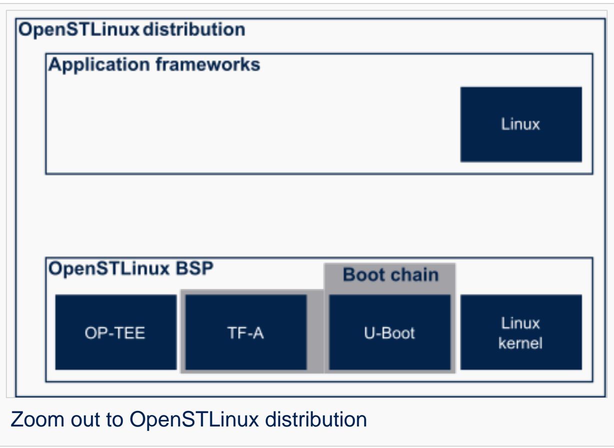
Contents

1 Overview of the OP-TEE open source project 20
2 Architecture 21
- 2.1 OP-TEE core 21
- 2.2 OP-TEE trusted libraries 21
- 2.3 TEE Linux driver 22
- 2.4 TEE Client API 22
- 2.5 TEE supplicant 22
- 2.6 Host tools 22

3 Booting with OP-TEE 23
4 Invoking the OP-TEE services from Linux based OS 24
5 Experiencing OP-TEE on a target 25
6 References 26

1 Overview of the OP-TEE open source project

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 .



Zoom out to OpenSTLinux distribution

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.

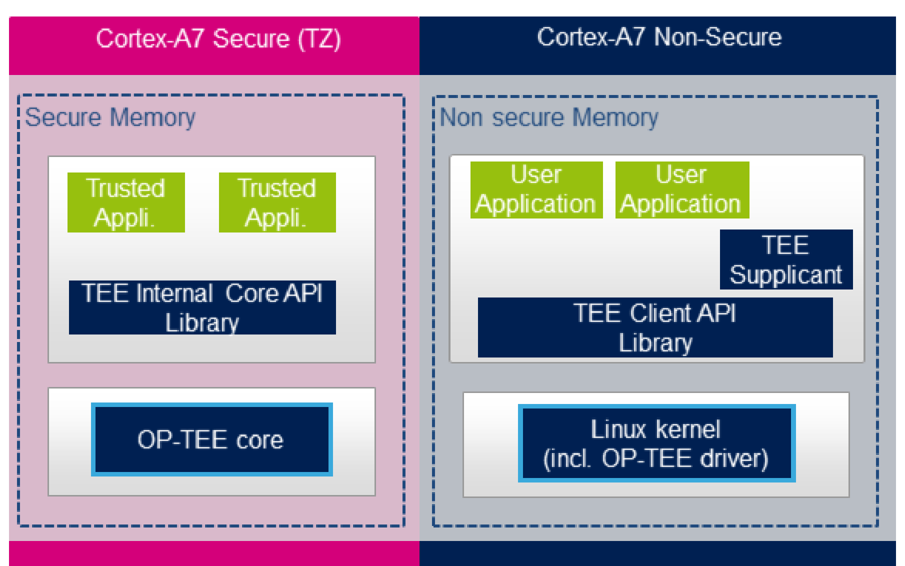
- OP-TEE official site[1]
- OP-TEE source repositories [2][3][4]
- OP-TEE documentation[5]

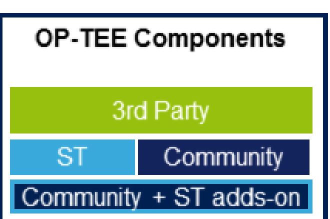
GlobalPlatform Device TEE specifications (TEE Client API, TEE Internal Core API and few more) is available from the GlobalPlatform site[6].

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 Restriced – 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.

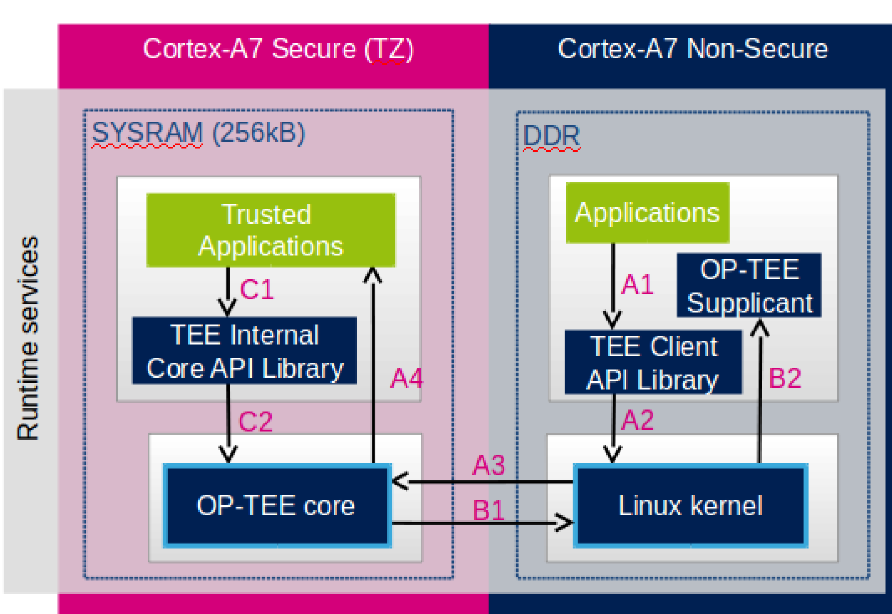
Refer to the target system boot sequences for more details.

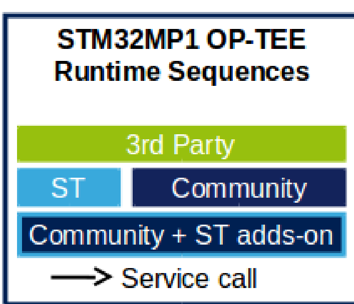
ST Restriced – Subject to Non-Disclosure Agreement – Do not copy

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.

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
```

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

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/

Open Portable Trusted Execution Environment

Linux® is a registered trademark of Linus Torvalds.

Operating System

TrustZone®

Arm® and TrustZone® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Trusted Execution Environment

Application programming interface

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. arm

Cortex®

Device Tree

Stable: 25.09.2020 - 09:10 / Revision: 25.09.2020 - 09:09

A quality version of this page, approved on *25 September 2020*, was based off this revision.

Contents

1 Article purpose 27
2 Peripheral overview 28
2.1 Features 28
2.2 Security support 28
3 Peripheral usage and associated software 29
3.1 Boot time 29
3.2 Runtime 29
3.2.1 Overview 29
3.2.2 Software frameworks 29
3.2.3 Peripheral configuration 30
3.2.4 Peripheral assignment 30
4 How to go further 31
5 References 32

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

1 Article purpose

The purpose of this article is to:

- briefly introduce the RCC peripheral and its main features
- indicate the level of security supported by this hardware block
- explain, when necessary, how to configure the RCC peripheral.

2 Peripheral overview

The **RCC** peripheral is used to control the internal peripherals, as well as the **reset** signals and **clock** distribution. The RCC gets several internal (LSI, HSI and CSI) and external (LSE and HSE) clocks. They are used as clock sources for the hardware blocks, either directly or indirectly, via the four PLLs (PLL1, PLL2, PLL3 and PLL4) that allow to achieve high frequencies.

2.1 Features

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

2.2 Security support

The RCC is a **secure** peripheral. There are two levels of security, which are controlled via two bits in the RCC_TZCR register (only accessible in secure mode):

- **TZEN** allows to set some RCC registers in secure mode, in particular registers for configuring PLL1 and PLL2, in order to secure a TrustZone perimeter for the Cortex®-A7 secure core and its peripherals.
- **MCKPROT** allows extending the TZEN secure clock control perimeter to PLL3 and to the MCU subsystem, so to the Cortex®-M4 and its bus clock.

Please note that all RCC registers can be read from the non-secure world.

3 Peripheral usage and associated software

3.1 Boot time

The RCC security level differs for each boot chain:

- the trusted boot chain sets TZEN to 1 an MCKPROT to 0
- the basic boot chain sets TZEN to 0 an MCKPROT to 0

The RCC is used by all the boot components: the ROM code, the FSBL, the SSBL and up to the Linux® kernel. Nevertheless, the main initialization step is performed by the FSBL that is responsible for the clock tree initialization: it consists in configuring all the input clocks, the PLL and the clock sources that are selected as kernel clocks for all peripherals. The whole configuration is carried out by the device tree.

The STM32CubeMX tool allows configuring in one place the clock tree that will be applied at boot time and used at runtime, so it is highly recommended to use it to generate your device tree. Moreover, the STM32CubeMX integrates all the information documented in the STM32MP15 reference manuals, making this configuration step straighforward.

3.2 Runtime

3.2.1 Overview

The RCC peripheral is shared at runtime:

- the Arm®Cortex®-A7 secure core controls all the secure registers (refer to TZEN and MCKPROT bit descriptions) through the RCC OP-TEE driver. The access to some secure registers from the Cortex®-A7 non-secure core can be achieved via runtime secure services implemented in the secure monitor (from the OP-TEE if it is present, otherwise from the TF-A).
- the Arm®Cortex®-A7 non-secure core controls the clock management via the clock framework, and the reset management via the reset framework in Linux®.
- the Arm®Cortex®-M4 core controls all the clock and reset managements in STM32Cube with the RCC HAL driver

Concurrent control from each context is possible because the above managements are performed via independent registers.

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
Power & Thermal	RCC	OP-TEE RCC driver	Reset framework Clock framework	STM32Cube RCC driver	

3.2.3 Peripheral configuration

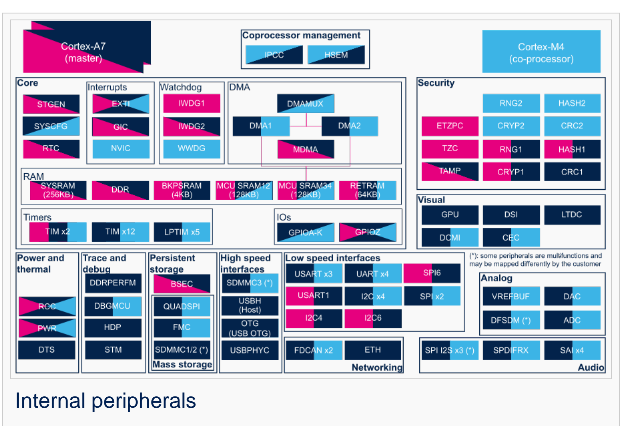
The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration can be done alone via the STM32CubeMX tool for all internal peripherals, and then manually completed (particularly for external peripherals), according to the information given in the corresponding software framework 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.

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
.



Internal peripherals

Domain	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
Power & Thermal	RCC	RCC				

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

4 How to go further

The RCC is interfaced with the HDP internal peripheral, thus offering the flexibility to monitor the main RCC state signals on the debug pins.

Please refer to the STM32MP15 reference manuals for the full list of signals that can be monitored.

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

5 References

Reset and Clock Control

Low Speed Internal oscillator (STM32 clock source)

High Speed Internal oscillator (STM32 clock source) or High Speed Synchronous Serial Interface (MIPI® Alliance standard)

Multi Speed Internal oscillator (STM32 clock source)

Low Speed External oscillator (STM32 clock source)

High Speed External oscillator (STM32 clock source)

TrustZone®

Arm® and TrustZone® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Cortex®

Microcontroller Unit (MCUs have internal flash memory and are intended to operate with a minimum amount of external support ICs. They commonly are a self-contained, system-on-chip (SoC) designs.)

Read Only Memory

First Stage Boot Loader

Second Stage Boot Loader

Linux® is a registered trademark of Linus Torvalds.

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Open Portable Trusted Execution Environment

Stable: 20.07.2021 - 09:02 / Revision: 11.03.2021 - 08:07

A quality version of this page, approved on *20 July 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.

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	NEW 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	v1.0
AN5225	USB Type-C™ Power Delivery using STM32xx Series MCUs and STM32xxx Series MPUs	AN5225.pdf	v3.0
AN5253	Migration of microcontroller applications from STM32F4x9 lines to STM32MP151, STM32MP153 and STM32MP157 lines microprocessor	AN5253.pdf	v1.0
AN5256	STM32MP151, STM32MP153 and STM32MP157 discrete power supply hardware integration	AN5256.pdf	v2.0
AN5260	STM32MP151/153/157 MPU lines and STPMIC1B integration on a battery powered application	AN5260.pdf	NEW v2.0
AN5275	USB DFU/USART protocols used in STM32MP1 Series bootloaders	AN5275.pdf	v1.0
AN5284	STM32MP1 series system power consumption	AN5284.pdf	v1.0
AN5348	FDCAN peripheral on STM32 devices	AN5348.pdf	v1.0
AN5431	The STPMIC1 PCB layout guidlines	AN5431.pdf	v1.0
AN5438	STM32MP1 Series lifetime estimates	AN5438.pdf	v1.0
	Overview of the secure secret provisioning (SSP) on STM32MP1	AN551	

Reference	Name	Link	Version
Application notes			
AN5510	Series	0.pdf	v1.0
Datasheets[1]			
DS12505	STM32MP157C/F datasheet (secure)	DS12505.pdf	NEW v5.0
DS12504	STM32MP157A/D datasheet (basic)	DS12504.pdf	NEW v5.0
DS12503	STM32MP153C/F datasheet (secure)	DS12503.pdf	NEW v5.0
DS12502	STM32MP153A/D datasheet (basic)	DS12502.pdf	NEW v5.0
DS12501	STM32MP151C/F datasheet (secure)	DS12501.pdf	NEW v5.0
DS12500	STM32MP151A/D datasheet (basic)	DS12500.pdf	NEW v5.0
DS12792	STPMIC1 datasheet	DS12792.pdf	NEW v7.0
Errata sheets			
ES0438	STM32MP15xx device errata	ES0438.pdf	NEW v6.0
Reference manuals[1]			
RM0436	STM32MP157 reference manual (STM32MP157xxx advanced Arm®-based 32-bit MPUs)	RM0436.pdf	NEW v5.0
RM0442	STM32MP153 reference manual (STM32MP153xxx advanced Arm®-based 32-bit MPUs)	RM0442.pdf	NEW v5.0
RM0441	STM32MP151 reference manual (STM32MP151xxx advanced Arm®-based 32-bit MPUs)	RM0441.pdf	NEW 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

Reference	Name	Link	Version
Application notes			
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 v2.0
UM2579	Migration guide from System Workbench to STM32CubeIDE	UM2579.pdf	v1.0
UM2553	STM32CubeIDE quick start guide	UM2553.pdf	NEW v2.0
AN5360	Getting started with projects based on the STM32MP1 Series in STM32CubeIDE	AN5360.pdf	v1.0
UM2609	STM32CubeIDE user guide	UM2609.pdf	NEW v3.0
UM1718	STM32CubeMX user manual	UM1718.pdf	NEW v34.0
UM2237	STM32CubeProgrammer tool user manual	UM2237.pdf	NEW v15.0
UM2238	STM32 Trusted Package Creator tool user manual	UM2238.pdf	v7.0
UM2542	STM32 Series Key Generator tool user manual	UM2542.pdf	v1.0
UM2543	STM32 Series Signing tool user manual	UM2543.pdf	v1.0

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

- 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-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)

Doubledata rate (memory domain)

USB port or connector

Microprocessor Unit

Device Firmware Upgrade

Universal Synchronous/Asynchronous Receiver/Transmitter

Printed Circuit Board

Secure Secret Provisioning

Secure secrets provisioning

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Display Serial Interface (MIPI® Alliance standard)

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 39
2 Architecture 40
3 Boot loader stages 42
3.1 BL1 42
3.2 BL2 42
3.2.1 FIP 42
3.2.2 Firmware Configuration 42
3.2.3 Authentication 43

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

3.3 BL32 43
4 References 44

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]

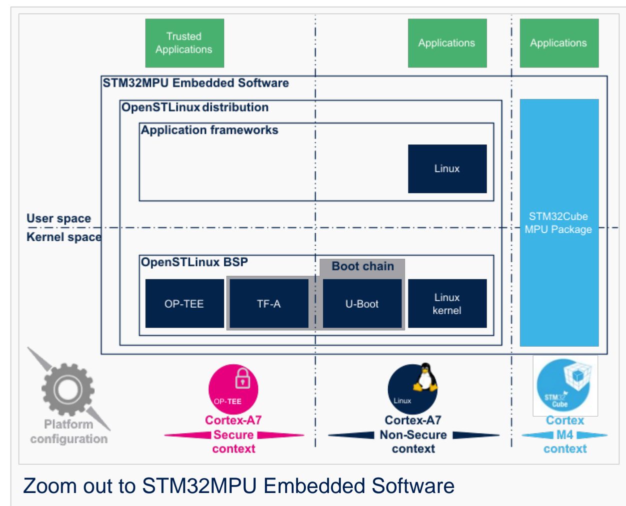
It is used as the first-stage boot loader (FSBL) on STM32 MPU platforms when using the trusted boot chain.

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

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):

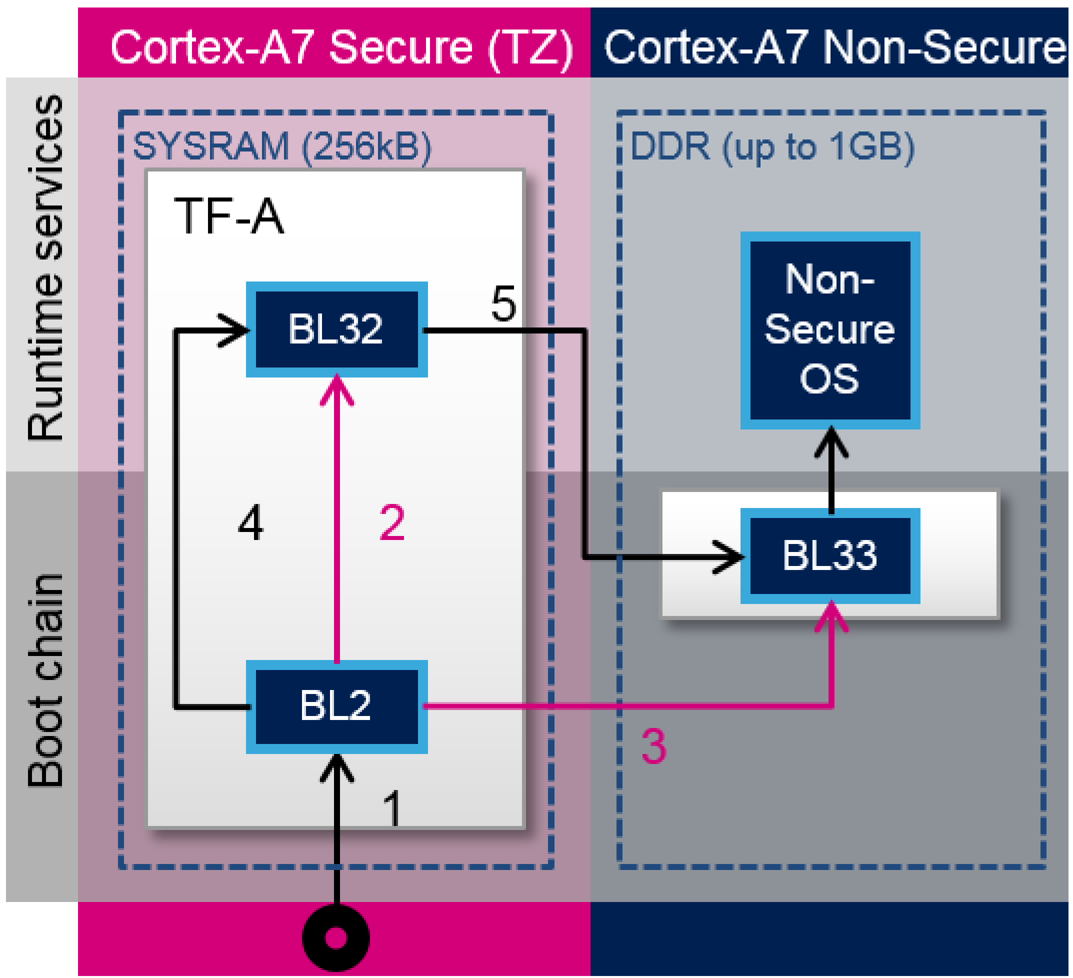
- Boot loader stage 1 (BL1) application processor trusted ROM
- Boot loader stage 2 (BL2) trusted boot firmware
- Boot loader stage 3-2 (BL32) runtime software
- Boot loader stage 3-3 (BL33) non-trusted firmware

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

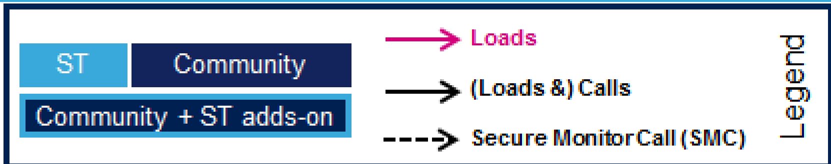
Because STM32 MPU platforms uses a dedicated ROM code, the BL1 boot stage is then removed. ROM code expects the BL2 to run at EL3 execution level. This mode is selected when the BL2_AT_EL3 build flag is enabled.

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.



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



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).

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

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

First Stage Boot Loader

Microprocessor Unit

Power State Coordination Interface

Secure Monitor Call

Trusted Firmware for Arm® Cortex®-A

Boot Loader stage 1

Read Only Memory

Boot Loader stage 2

Boot Loader stage 3-2

Boot Loader stage 3-3

Second Stage Boot Loader

Linux® is a registered trademark of Linus Torvalds.

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Doubledata rate (memory domain)

Firmware Image Package is a packaging format used by TF-A

Firmware Configuration Framework used by TF-A - NEW

Operating System

Chain of Trust

Secure coprocessor

Secure Payload minimal

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

Trusted Execution Environment

Open Portable Trusted Execution Environment

System control and management interface

Stable: 30.03.2021 - 13:21 / Revision: 29.03.2021 - 09:46

A quality version of this page, approved on *30 March 2021*, was based off this revision.

Contents

1 Article purpose ... 46
2 Peripheral overview ... 47
2.1 Features ... 47
2.2 Security support ... 47
3 Peripheral usage and associated software ... 48
3.1 Boot time ... 48
3.2 Runtime ... 48
3.2.1 Overview ... 48
3.2.2 Software frameworks ... 48
3.2.3 Peripheral configuration ... 48
3.2.4 Peripheral assignment ... 48
4 How to go further ... 51
5 References ... 52

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

1 Article purpose

The purpose of this article is to

- briefly introduce the **TIM** 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 how to configure the TIM peripheral

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

2 Peripheral overview

The TIM peripheral is a multi-channel timer unit, available in various configurations, depending on the instance used. There are basically following categories: advanced-control timers, general-purpose timers and basic timers.

The TIM can provide: PWM with complementary output and dead-time insertion, break detection, input capture[1], quadrature encoder[2] interface (typically used for rotary encoders), trigger source for other internal peripherals like: ADC[3], DAC[4], DFSDM[5].

2.1 Features

The **TIM** peripheral is available in different configurations, depending on the selected instance :

- TIM1 and TIM8 are advanced-control timers, with 6 independent channels.
- TIM2, TIM3, TIM4 and TIM5 are general-purpose timers, with 4 independent channels.
- TIM12, TIM13 and TIM14 are general-purpose timers, with 2 (TIM12) or 1 (TIM13 and TIM14) independent channels.
- TIM15, TIM16 and TIM17 are also general-purpose timers, with 2 (TIM15) or 1 (TIM16 and TIM17) independent channels.

Compare to TIM12, TIM13 and TIM14, this configuration brings some features that are very useful for motor control (like break function, DMA burst mode control, complementary output with dead-time insertion, ...)

- TIM6 and TIM7 are basic timers

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

2.2 Security support

The TIM is a **non-secure** peripheral.

3 Peripheral usage and associated software

3.1 Boot time

The TIM is not used at boot time.

3.2 Runtime

3.2.1 Overview

TIM12 and/or **TIM15** can be allocated to:

- the Arm®Cortex®-A7 secure core to be controlled in the secure monitor (TF-A or OP-TEE), to perform HSI and CSI calibrations[6] in RCC.

All TIM instances can be allocated to:

- the Arm®Cortex®-A7 non-secure to be controlled in Linux® by the PWM, the IIO, and/or the *Counter* frameworks.

or

- the Arm®Cortex®-M4 to be controlled in STM32Cube MPU Package by TIM HAL driver

Note that RCC[7] owns one prescaler per **TIM group** corresponding to **APB1** and **APB2** buses: TIMG1PRE and TIMG2PRE, respectively. The allocation to Cortex-A7 or the Cortex-M4 should ideally be done on a per group basis to get independent clocking setup on each side, this is why the TIM instances groups are shown in the summary table below (#Peripheral assignment).

3.2.2 Software frameworks

Domain	Peripheral	Software components			Comment
OP-TEE	Linux	STM32Cube			
Core/Timers	TIM	TF-A TIM driver OP-TEE TIM driver	PWM framework IIO framework, *Counter* framework	STM32Cube TIM driver	

3.2.3 Peripheral configuration

The configuration is applied by the firmware running in the context to which the peripheral is assigned. The configuration by itself can be performed via the STM32CubeMX tool for all internal peripherals. It can then be manually completed (especially for external peripherals) according to the information given in the corresponding software framework article.

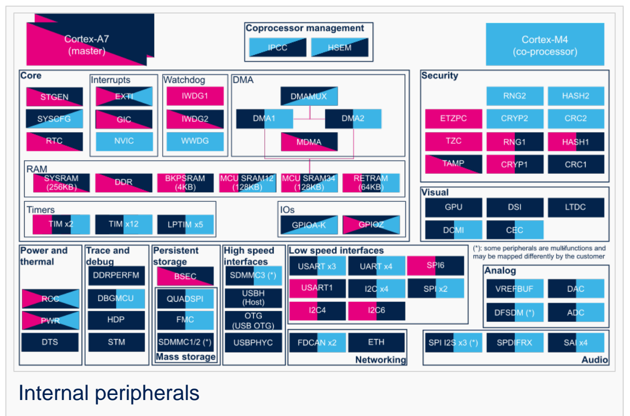
For Linux kernel configuration, please refer to TIM device tree configuration and TIM Linux driver articles.

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.

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
.



Internal peripherals

Domain	Peripheral	Runtime allocation				Comment
Instance	Cortex-A7 secure (OP-TEE)	Cortex-A7 non-secure (Linux)	Cortex-M4 (STM32Cube)			
		TIM1 (APB2 group)				Assignment (single choice)
		TIM2 (APB1 group)				Assignment (single choice)
		TIM3 (APB1 group)				Assignment (single choice)
		TIM4 (APB1 group)				Assignment (single choice)
		TIM5 (APB1 group)				Assignment (single choice)
		TIM6 (APB1 group)				Assignment (single choice)
		TIM7 (APB1 group)				Assignment (single choice)
Core/Timers	TIM	TIM8				Assignment (single

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

Domain	Peripheral	Runtime allocation				Comment
		(APB2 group)				choice)
		TIM12 (APB1 group)				Assignment (single choice)
		TIM13 (APB1 group)				Assignment (single choice)
		TIM14 (APB1 group)				Assignment (single choice)
		TIM15 (APB2 group)				Assignment (single choice)
		TIM16 (APB2 group)				Assignment (single choice)
		TIM17 (APB2 group)				Assignment (single choice)

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

4 How to go further

STM32 cross-series timer overview[8] application note.

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

5 References

- Input capture
- Quadrature encoder
- ADC internal peripheral
- DAC internal peripheral
- DFSDM internal peripheral
- How to activate HSI and CSI oscillators calibration
- RCC internal peripheral
- STM32 cross-series timer overview application note

Pulse Width Modulation

Analog-to-digital converter. The process of converting a sampled analog signal to a digital code that represents the amplitude of the original signal sample.

Digital-to-analog converter (Electronic circuit that converts a binary number into a continuously varying value.)

Digital Filter for Sigma-Delta Modulator

Direct Memory Access

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. arm

Cortex®

High Speed Internal oscillator (STM32 clock source) or High Speed Synchronous Serial Interface (MIPI® Alliance standard)

Multi Speed Internal oscillator (STM32 clock source)

Linux® is a registered trademark of Linus Torvalds.

Microprocessor Unit

Reset and Clock Control

Open Portable Trusted Execution Environment