



Welcome on this training describing STM32 MPU Embedded Software architecture for STM32MP1 series.

This presentation aims to introduce the architecture for the various embedded software components, to show how the platform is booting and to give pointers to the STM32 MPU Wiki, where you will be able to find up to date and more detailed information.

Agenda

- 1 STM32 MPU concepts & applicability management
- 2 STM32 MPU peripherals overview
- 3 STM32 MPU embedded software
- 4 STM32MP1 series boot chain
- 5 STM32MP1 series device tree
- 6 STM32MP1 series flash mapping



2


Here is the agenda of this online training:

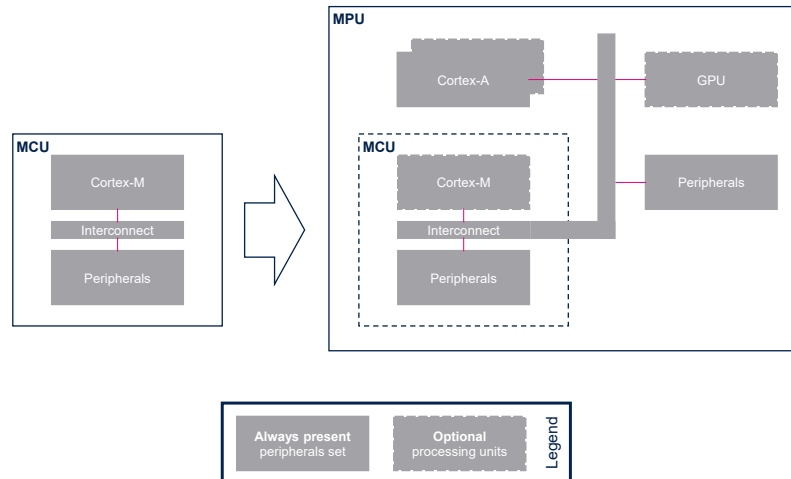
- First, we will explain few concepts required for a proper understanding of the presentation,
- Then, we will introduce the peripherals embedded in STM32MP1 series and see how they can be controlled by the software,
- This will lead us to dig into the software components delivered by STMicroelectronics to support those series
- After this static view, we will introduce some dynamics and explain the boot chain, including security aspects
- The device tree is widely used for software components configuration, and we will see how the device tree source files are organized
- Then, an overview of the typical flash memory mapping will be given

STM32 MPU concepts & applicability



From MCU to MPU

 [Getting started with STM32 MPU devices]

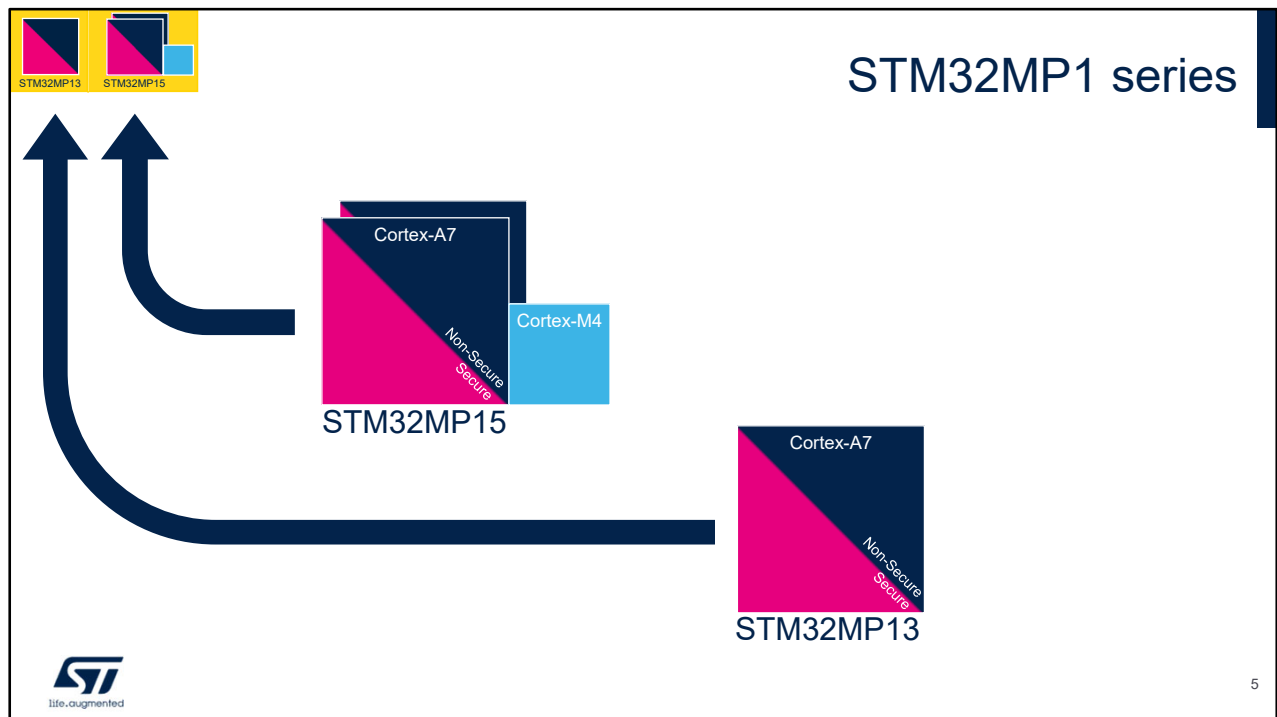


4

Microcontroller units, or MCUs, are built around cores such as the Arm Cortex-M. Such cores are very efficient for deterministic operations in a bare metal or real-time operating system context.

Microprocessor units, or MPUs, rely on cores such as the Arm Cortex-A, with a memory management unit to manage virtual memory spaces, opening the door to efficient support of rich operating systems such as Linux. A fast interconnect provides a bridge between the processing unit, high-bandwidth peripherals, external memories and, usually, a graphical Processing Unit.

This figure also shows that it is conceptually possible to embed an MCU sub-system, with an Arm Cortex-M, inside an MPU, leading to what is called a heterogeneous architecture.



The STM32MP1 microprocessor series is built around Arm Cortex-A7 and Cortex-M4 cores:

- The STM32MP15 is a heterogenous architecture which embeds up to two Cortex-A7s and one Cortex-M4
- The STM32MP13 embeds a single Cortex-A7

In this presentation, the icons in the top left corner will be used to show whether the current slide is applicable to one or both part numbers.

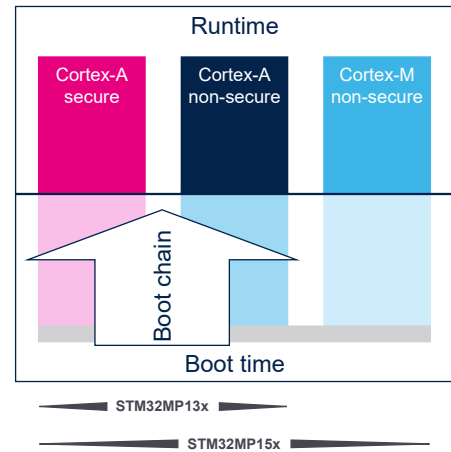
When there are differences to be highlighted, the STM32MP15 will be shown first, since it was the first part number launched within the STM32MP1 family, and the STM32MP13 changes will be highlighted just after.



Runtime context terminology

[Getting started with STM32 MPU devices]

- Hardware execution context
 - « a core and a security mode »
- Firmware executed in runtime context
 - Arm Cortex-A secure (TrustZone) executes OP-TEE
 - Arm Cortex-A non secure executes Linux
 - Arm Cortex-M (non-secure) executes STM32Cube
- Peripheral assignment to the runtime contexts
 - Assigned or shared



6

The Arm Cortex-A7 core owns two hardware execution contexts, corresponding to the non-secure and secure modes.

The Arm Cortex-M4 core only has one hardware execution context, which is non-secure.

Each hardware execution context executes a given firmware at runtime.

For instance, Linux is running in the Cortex-A7 non-secure context.

Each firmware can use two kinds of peripherals:

- assigned peripherals which will be exclusively used by the firmware itself,
- and shared peripherals which will be used by the firmware but also by one or more other firmware running in other contexts.



Peripherals assignment via STM32CubeMX



Categories	A->Z				
	Boot ROM	Boot loader	A7S (OP-TEE)	A7NS (Linux)	Cortex-M4 (Stm32Cube)
TIM6				<input checked="" type="checkbox"/>	<input type="checkbox"/>
TIM7				<input type="checkbox"/>	<input checked="" type="checkbox"/>
TIM8				<input checked="" type="checkbox"/>	<input type="checkbox"/>
TIM12			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
TIM13				<input type="checkbox"/>	<input type="checkbox"/>
TIM14				<input type="checkbox"/>	<input type="checkbox"/>

STM32MP13x

STM32MP15x



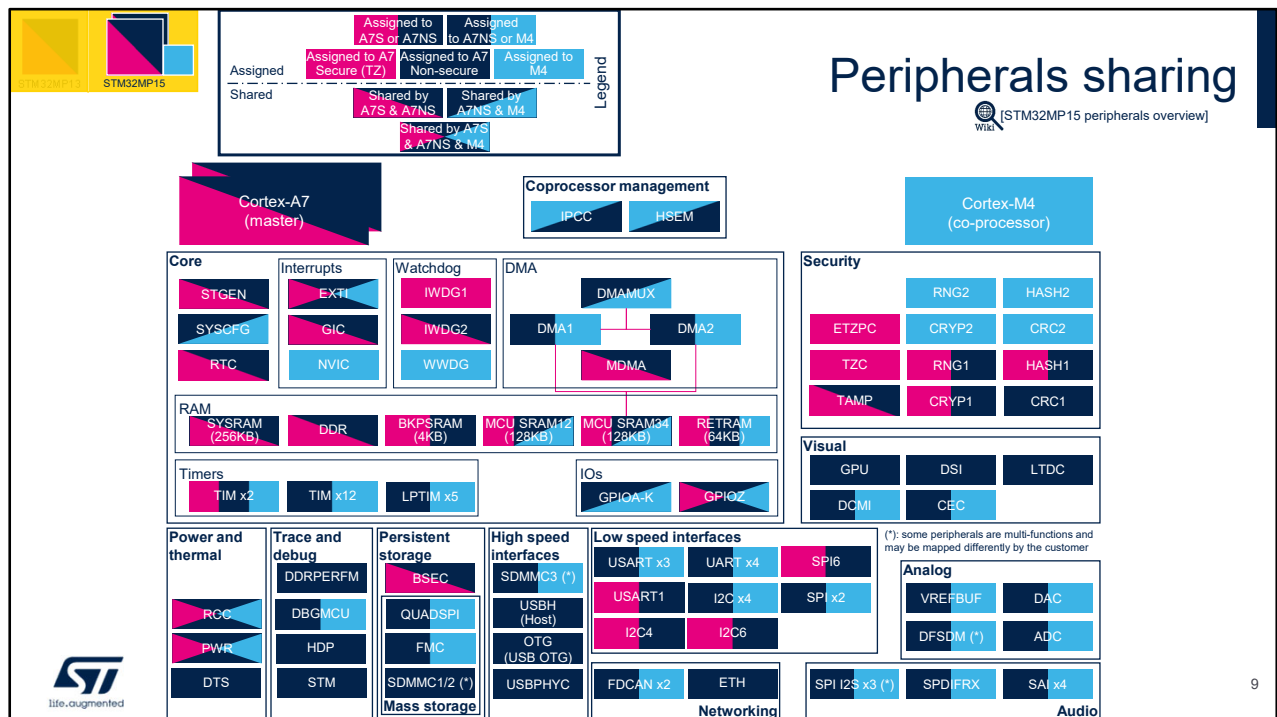
7

Note that, among other things, the STM32CubeMX tool can be used to assign the peripherals to the runtime contexts and generate the firmware initialization code that will put the platform firewalling in place.

For instance, a peripheral assigned to the Cortex-A7 secure context will physically be isolated and not be accessible from the Cortex-A7 non-secure context.

STM32 MPU peripherals overview





This diagram shows all STM32MP15 peripherals, grouped per functional domain and colored according to their assignment capabilities.

In line with the concepts introduced in the previous slide, the legend of this diagram distinguishes two possible states for each peripheral:

- “Assigned” means that only one hardware execution context is using the peripheral at runtime. A single-color box means that the assignment is static whereas a double or triple color box with vertical separator means that a user choice is needed to assign the peripheral to a given context, depending on its application needs. This assignment can be made with the STM32CubeMX tool, and it is reinforced by hardware isolation for the Cortex-A7 secure and for the Cortex-M4.

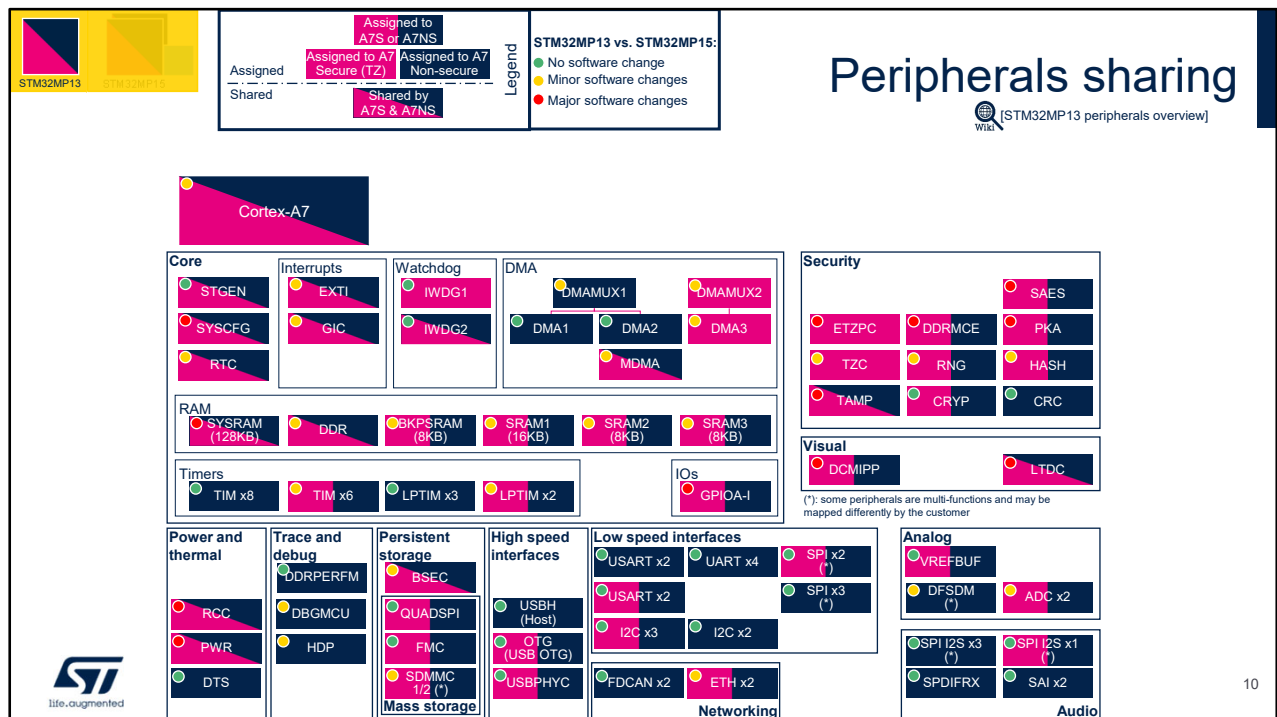
- “Shared” means that the peripheral can be concurrently used by two or even three different execution contexts. This mode implies registers banking or other mechanisms that will ensure there is no contention between the contexts when they access a common resource.

For instance:

- TIM instances, on the left, can be assigned to one runtime context and will only be used by this one
- RCC, just below, is a system peripheral that can be concurrently accessed by the three runtime contexts

All the mechanisms involved in the assignments are further described in STM32 MPU Wiki, starting from the [STM32MP15 peripherals overview] article.

Please refer to the reference manual for more information on all of these peripherals.



This diagram shows the peripheral assignment capabilities for STM32MP13, in the same way as the previous slide did ifor STM32MP15.

In addition, the green, yellow and red bullets identify the level of changes on STM32MP13, compared to STM32MP15.

Let's focus on the red ones, with major impacts on the software:

- SYSCFG is now secure-aware, to be consistent with all securable peripherals it allows to be configured, such as I2C or ethernet,
- Each GPIO pin is individually securable.
- The SYSRAM is reduced to 128 KB but this must be balanced with the introduction of the DDR encryption,

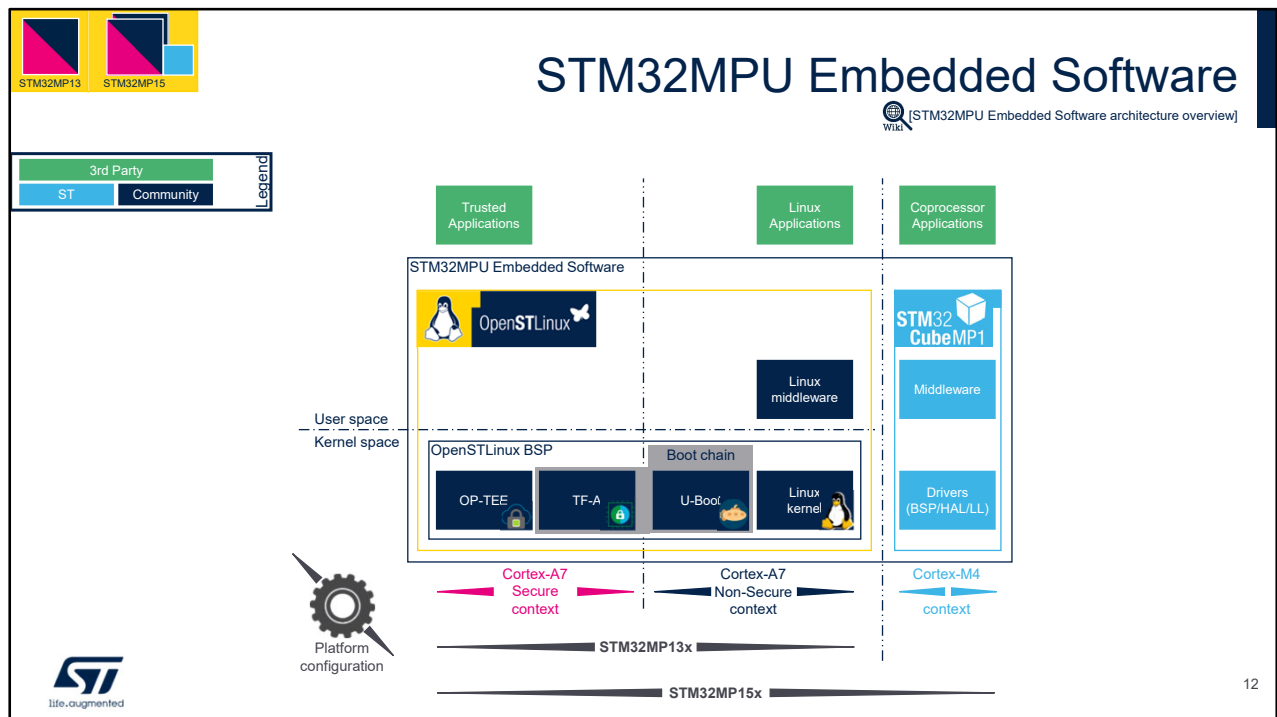
- performed by DDRMCE,
- Continuing with the security side:
 - SAES is a side-channel attack-protected symmetric cryptography accelerator,
 - PKA is a side-channel attack-protected accelerator for elliptic curve cryptographic operations,
 - TAMP has been enhanced with wider detection and reaction mechanisms,
 - ETZPC has been impacted because a larger number of peripherals are securable on STM32MP13, and this is the peripheral in charge of all internal memories and peripherals isolation
 - On the visual side, the DCMIPP is a new camera interface which comes with higher performance, and this is the same for LTDC on the display controller side, which is enhanced with a secure layer that can be filled in by the Cortex-A7 secure execution context, among other things.
 - RCC has been improved with finer grain security management,
 - PWR gets dedicated power domains for the Cortex-A7 supply and for SD card interfaces. Moreover, the STM32MP13 has a new low-power mode, called LPLV-Stop2, further described in the low power management online training.

All the mechanisms involved in the assignments are further described in STM32 MPU Wiki, starting from the [STM32MP13 peripherals overview] article.

Please refer to the reference manual for more information on all of these peripherals.

STM32 MPU embedded software



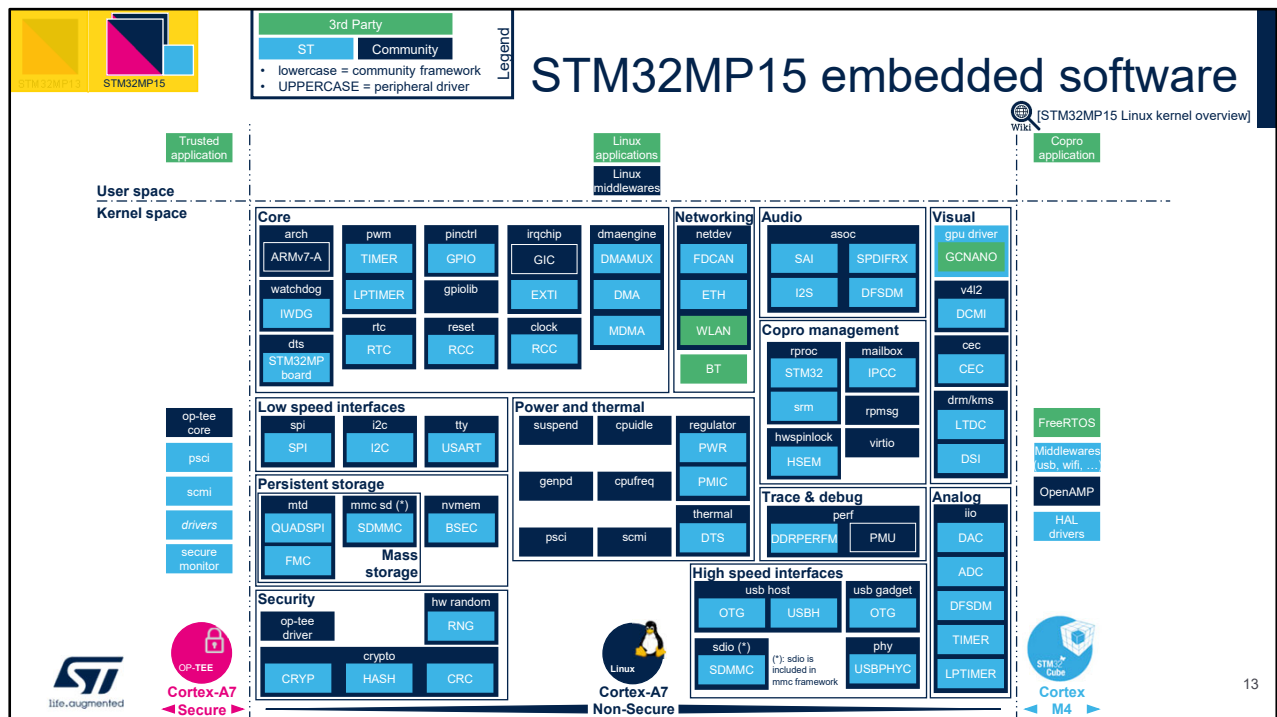


This diagram is very important to explain the terminology used to define STM32 MPU Embedded Software's main components:

- The OpenSTLinux distribution, running on the Arm® Cortex®-A, includes:
 - The OpenSTLinux board support package 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.
 - Linux middleware, relying on the BSP and providing API to run Applications that typically interact with the user via a display, a touchscreen, etc.

- The STM32CubeMP1 Package running on the Arm® Cortex®-M, only present on STM32MP15: it is based on HAL drivers and middleware, like other STM32 microcontrollers, completed with coprocessor management.

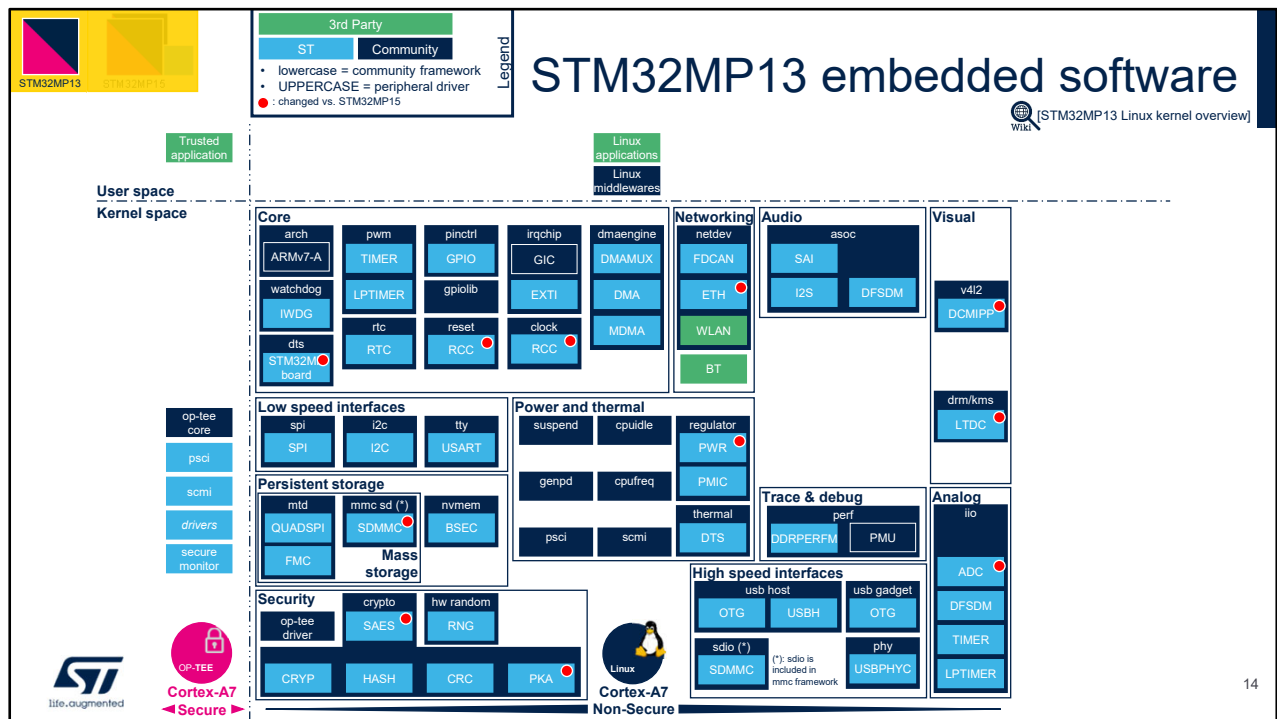
This diagram is available as a clickable image in STM32 MPU Wiki, allowing you to easily jump into the area that you want to explore and get more details about it.



The purpose of this slide is to show all the STM32MP15 peripheral drivers and their respective Linux frameworks:

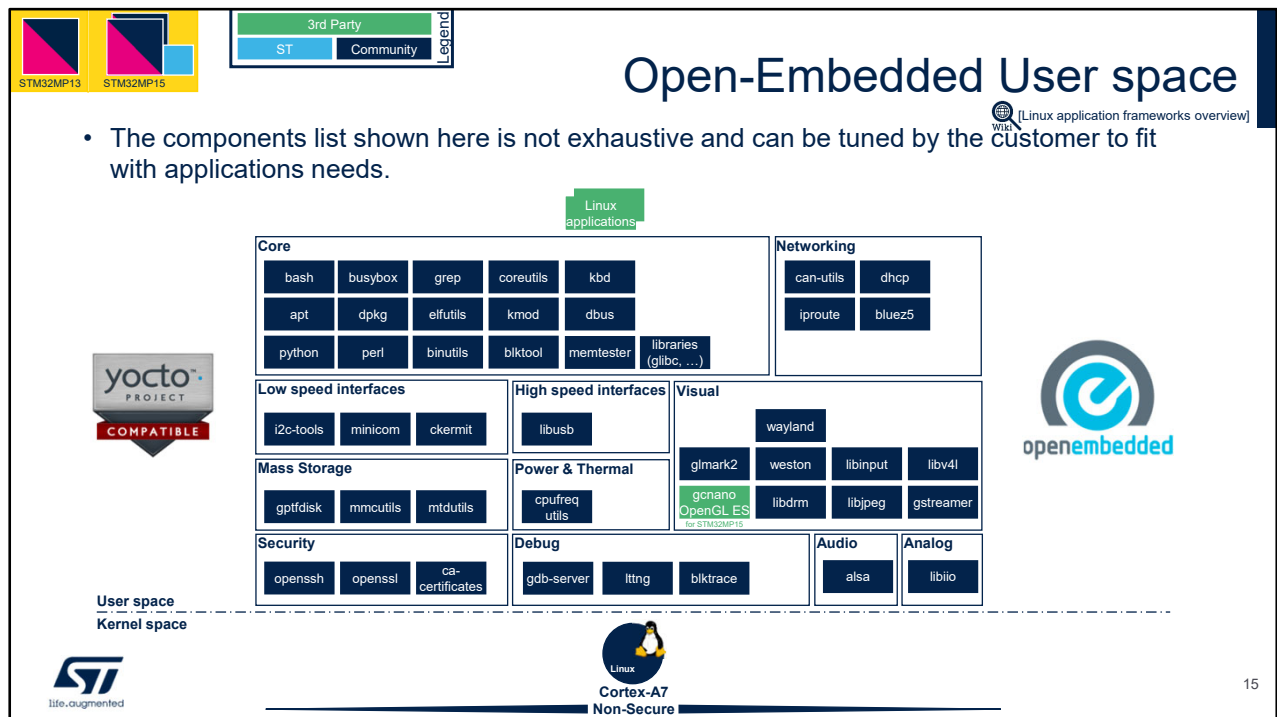
- the framework names are written in lowercase
- the peripheral driver names are written in uppercase
- the color code identifies each component source code origin: third party, community or STMicroelectronics

We will not dig into details in each box during this session: just keep in mind that this view exists in STM32 MPU Wiki and it allows you to delve into the various Linux frameworks and drivers.



Here is the same view for STM32MP13. The red bullets identify the peripherals that have led to the widest changes in Linux BSP.

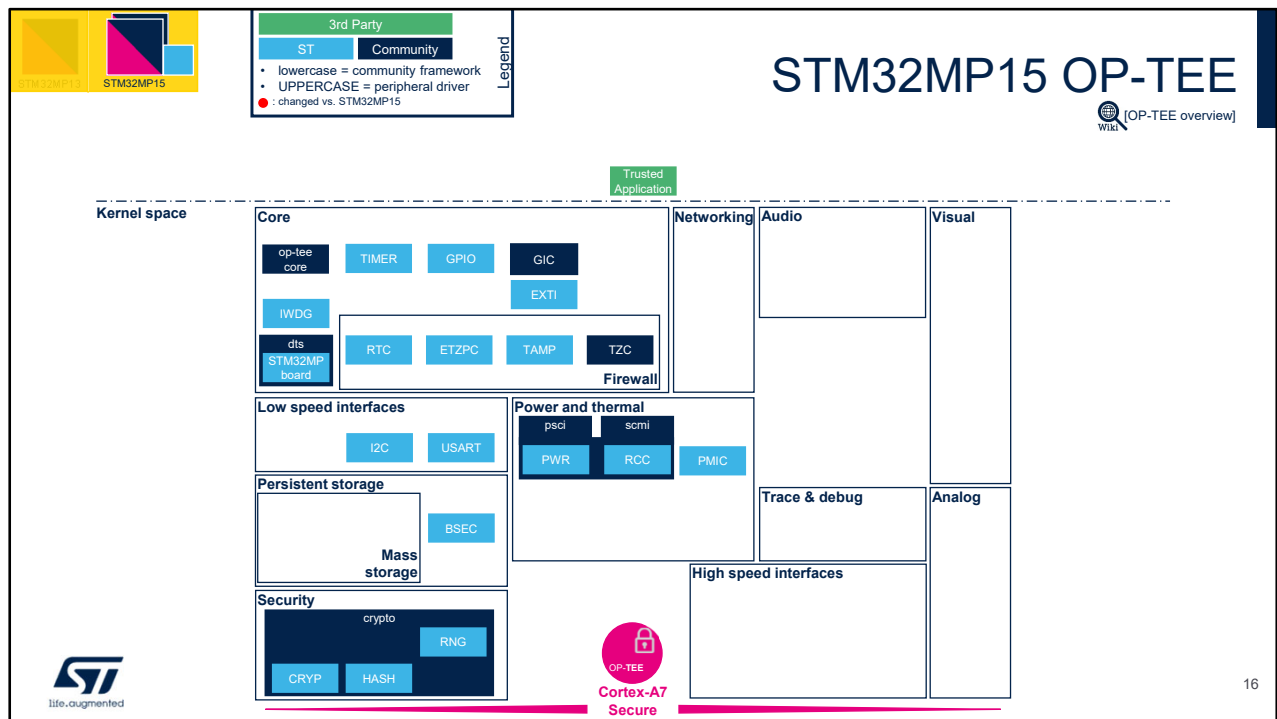
You can also find this diagram in STM32 MPU Wiki.



STM32MPU Embedded Software distribution is built on the Open-Embedded build framework for embedded Linux platforms.

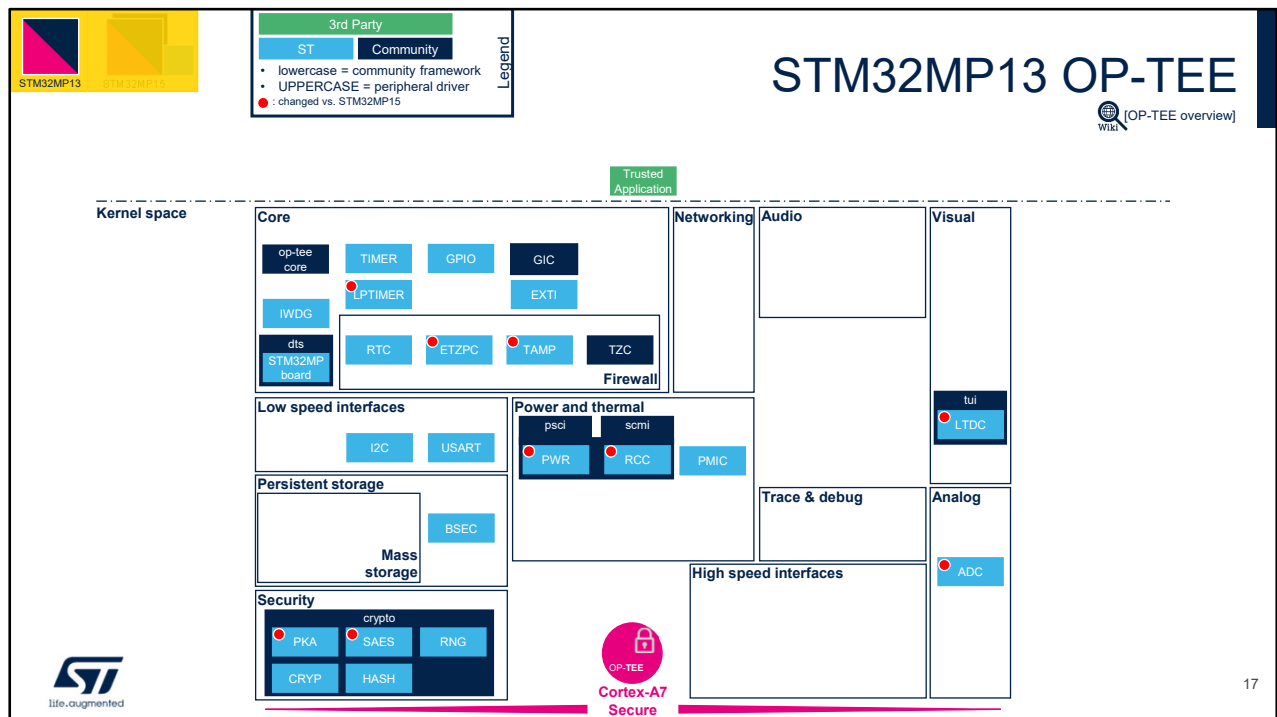
This diagram shows all of the conventional components embedded in our distribution root file system. Feel free to explore ST Wiki and Open-Embedded online documentation to customize this setup for your needs.

The GPU support binaries are present in the distribution for the whole STM32MP1 series, but only used while running on STM32MP15 since the STM32MP13 does not have any GPU.



Here is a detailed view of OP-TEE kernel, running on STM32MP15.

Due to the limited amount of SYSRAM in the circuit, the OP-TEE core pager is enabled in order to dynamically load and decrypt trusted application from the DDR to the SYSRAM at runtime.



Here is a detailed view of OP-TEE kernel, running on STM32MP13.

The main changes are due to the updating or addition of peripherals, as compared to STM32MP15, as explained in previous slides.

On STM32MP13, the DDR cyphering engine allows the external RAM to be trusted, so the OP-TEE can safely run from it, without requiring the use of the OP-TEE core pager.

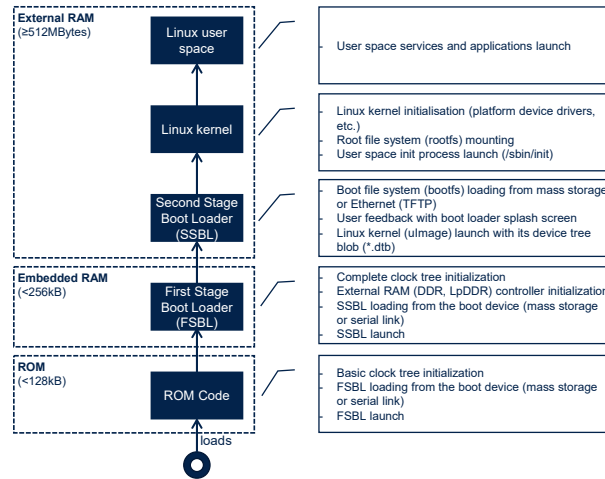
STM32MP1 series boot chain





Standard Linux boot chain

[Boot chain overview]



19

Booting a Linux platform is very similar to a rocket launch, since it is a multiple-stage process where only the last one is useful at the end. The boot chain shown here is standard in the sense that you may have already seen similar steps on other MPUs on the market.

The first stage is the ROM code: this is not a software component from a user perspective since this binary is embedded in the microprocessor and cannot be modified. The ROM code initializes a minimal clock tree in order to get all peripherals involved in the boot detection alive. Once this is done, the First Stage Boot Loader (or FSBL) is loaded from the selected boot device to the embedded RAM, then executed.

The FSBL is the first real software component executed, just

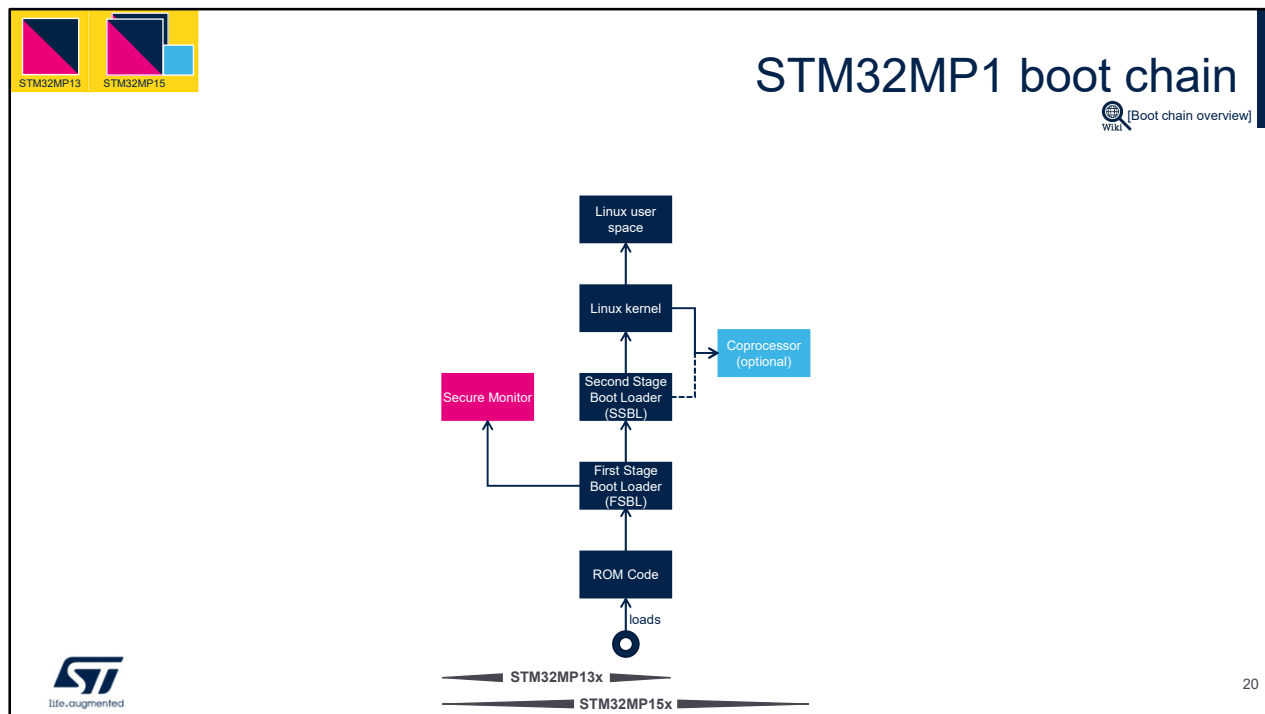
after the ROM code. It completes the clock tree initialization in order to get access to more peripherals with, among them, the external RAM where it loads the Second Stage Boot Loader (or SSBL) before execution.

The most famous SSBL used all around the world is U-Boot. Its mission is to load the Linux kernel into the external RAM from a given boot device among the ones supported by the ROM code or even other ones like the Ethernet, for instance. The SSBL has quite a large features set, and it is also often used to display an image during the start up process, which is called the splash screen. The “boot file system” or “bootfs” contains most of the binaries needed by U-Boot: The splash screen image, the Linux kernel and the device tree blob which contains all initialization data given to Linux kernel.

The last action from U-Boot is a jump to the Linux kernel entry point... and Linux is then alive! The kernel starts by initializing all its device drivers, then it mounts the “root file system” or “rootfs”, which contains all the user space applications and libraries.

The user space switch is realized when Linux kernel creates the “init” process, which launches the services and applications stored in the rootfs.

In dash lines boxes, you can see typical sizes of the successive memories embedding the boot chain components, from a few hundreds of kilo bytes for the internal memories up to several hundreds of mega bytes for the external memories.



Beyond Linux start-up, the STM32MP1 boot chain is also responsible for the start-up of two other major components of the processor:

- The secure monitor, supported by the Arm Cortex-A7 secure context, also called TrustZone®. Examples of the use of a secure monitor can be user authentication, key storage or tampering management.
- And the coprocessor firmware, running on the Arm Cortex-M4 core, which is only present on STM32MP15. It can be used to offload real time or low power services.

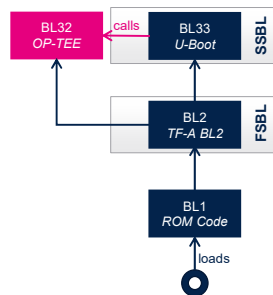
The dotted lines in the diagram mean that the STM32MP15 coprocessor can optionally be started by the second stage boot loader: we call this the “early boot”.



Trusted Firmware for Cortex-A terminology

WIKI [TF-A overview]

- <https://www.trustedfirmware.org/>
- BL = Boot Loader



21

Trusted Firmware community project uses the terminology shown here to identify the various boot loaders, so-called “BL”s:

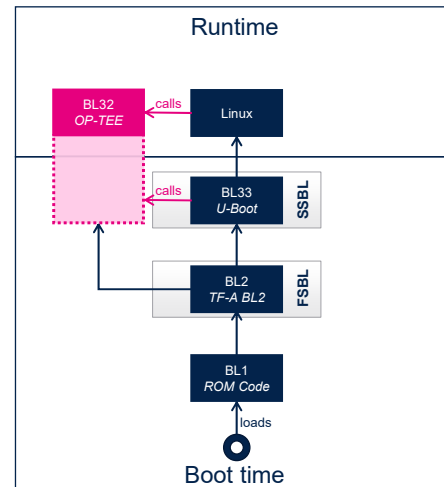
- BL1 can be assimilated to our ROM Code
- BL2 is implemented using TF-A BL2 source code
- BL32 is the secure monitor, such as the open-source OP-TEE implementation
- BL33 is the boot loader that starts Linux, i.e. U-Boot in our case. Note that U-Boot is running in the non-secure context and relies on secure monitor services whenever needed.



STM32MP1 OP-TEE runtime secure services

[TF-A overview]

- Platform system services in OP-TEE:
 - Power State Coordination Interface (PSCI)
 - i.e.: secondary cores, system power off, reboot...
 - System Control and Management Interface (SCMI)
 - i.e.: reset, clock and regulator controllers
 - Secure Monitor Call (SMC) proprietary services
 - i.e.: OTP access filtering
 - OP-TEE native services defined
 - by Global Platform TEE API specifications
 - or by services running in trusted applications (TA)



22

Beyond boot time, the figure on the right shows that OP-TEE remains resident at runtime, and is then able to answer Linux requests.

Various platform services are available in STM32MP1 OP-TEE implementation:

- PSCI services are used to control system states
- SCMI is used to expose filtered and abstracted access to critical system controls such as reset, clock and regulators
- SOC vendors or OEM can implement proprietary services, if needed
- On top of that, OP-TEE also implements native services answering to Global Platform specifications or supporting trusted applications

PSCI and SCMI use-cases are visible in the low power management online training.

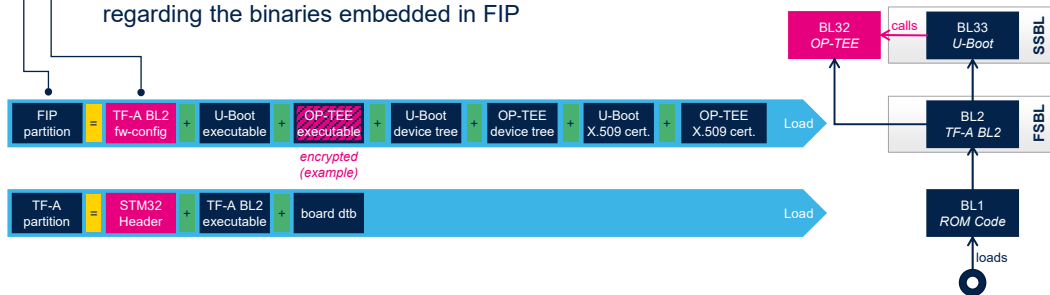


Trusted Firmware binaries packing

WIKI [TF-A overview]

- <https://www.trustedfirmware.org/>

- **FIP** = Firmware Image Package
 - A binary that is packing the various files loaded by TF-A BL2
- **FCONF** = Firmware Configuration (**fw-config**) Framework
 - A device tree used to configure **TF-A BL2** behavior, especially regarding the binaries embedded in FIP



23

Each boot stage is usually not loaded alone, as a binary that could be directly executed by the processor.

This is because some information and processing are needed before the real execution.

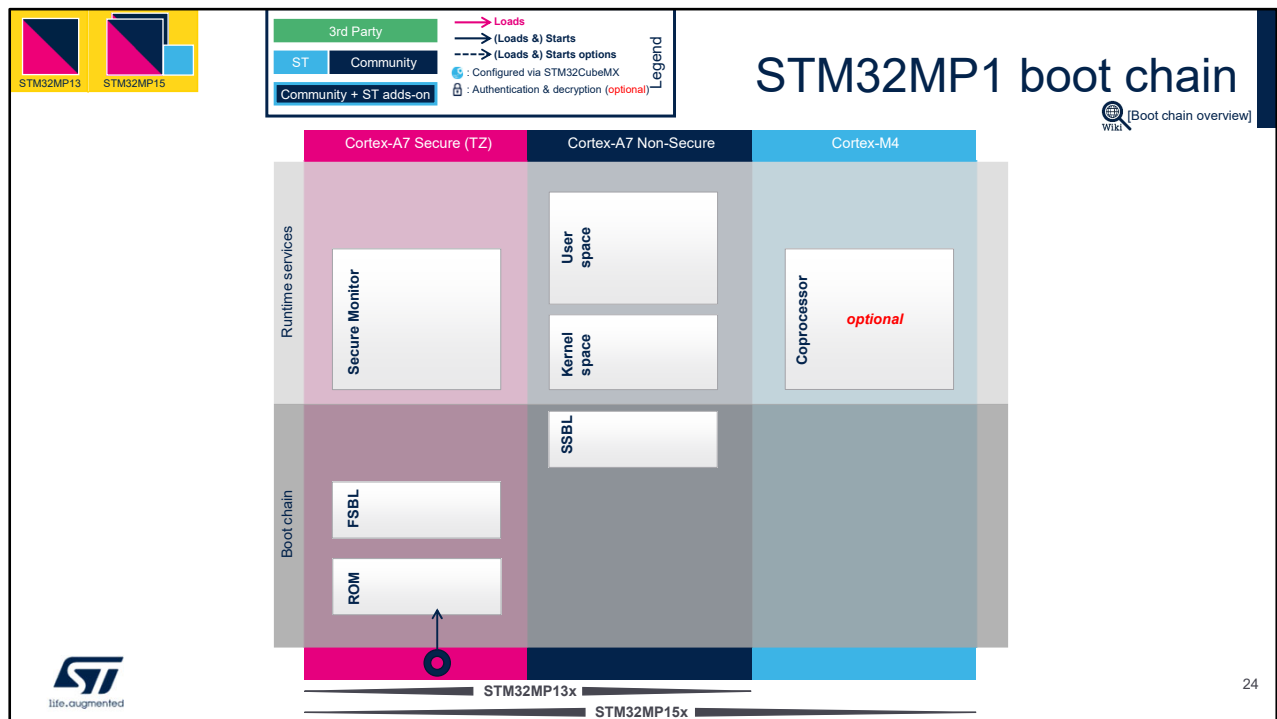
For instance, on the bottom, the ROM Code loads TF-A partition from an external flash or serial link, and this partition contains:

- A STM32 header which embeds information such as the binary size or authentication information, among other things
- TF-A binary which is itself split into several binaries: TF-A BL2 executable and the device tree blob file, reflecting the hardware board configuration.

During its execution, TF-A loads all the binaries it needs as a

single binary file, which is called the firmware image package or FIP:

- First, this pack embeds a firmware configuration file which will dictate to TF-A BL2 what to do with the various files embedded in the FIP
- Then, the FIP embeds various binaries, which can individually be encrypted (like OP-TEE, as example here) or authenticated (via X.509 certificates)



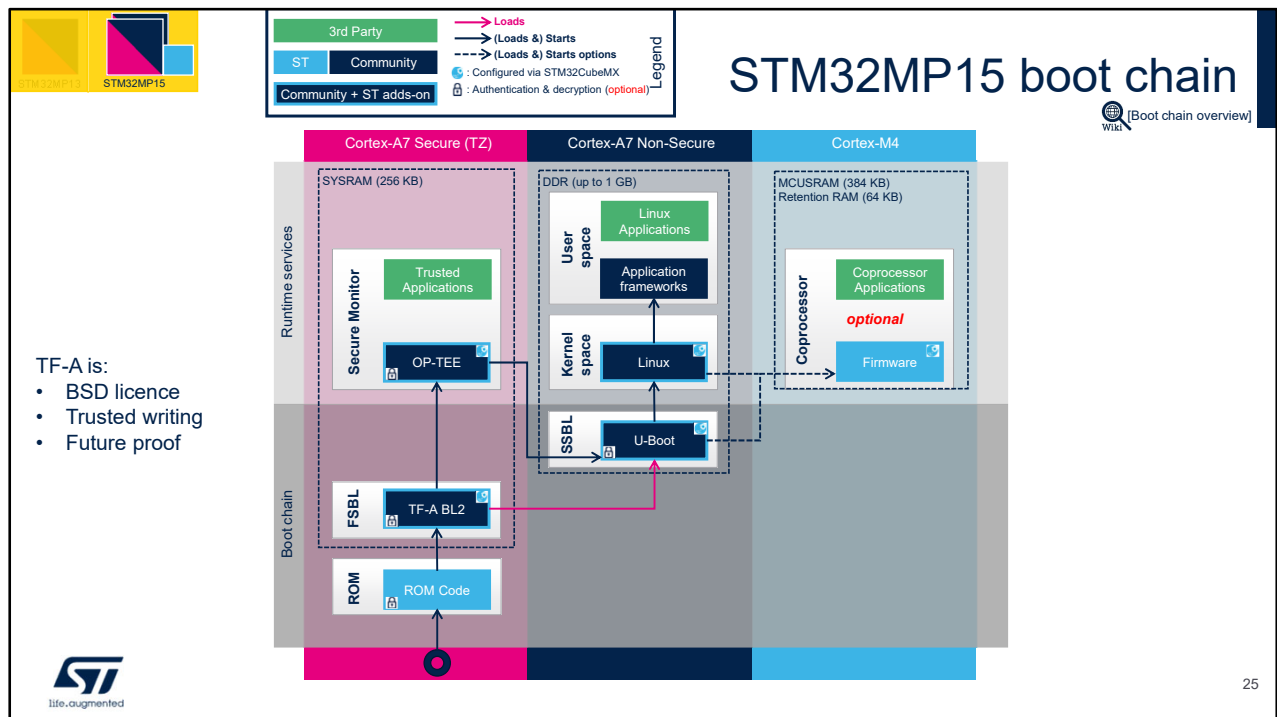
This diagram introduces, with colored vertical frames, the “hardware execution contexts” supported on the STM32MP1 platform:

- The Arm Cortex-A secure context, in pink
- The Arm Cortex-A non-secure context, in dark blue
- And the Arm Cortex-M context, in light blue, on STM32MP15 only

Grey horizontal frames show the boot chain in the bottom part and runtime services above.

Then, several boot stages, introduced in the previous slides, are mapped on the vertical and horizontal frames: the ROM code, the first stage boot loader, the second stage boot loader, the Linux kernel and user space. On top of that, the secure monitor is on the left and the coprocessor firmware

on the right.



The boot chain uses the Trusted Firmware for Cortex-A, also known as TF-A, as FSBL because:

- This bootloader is delivered under BSD license, which is sometimes preferred by customers who want to hide certain details or implementation in the boot chain,
- It was developed with the target of being trusted, so as to fulfil all requirements for customers who are sensitive to security problems,
- It is future-proof since it is widely used on the latest Arm architecture versions.

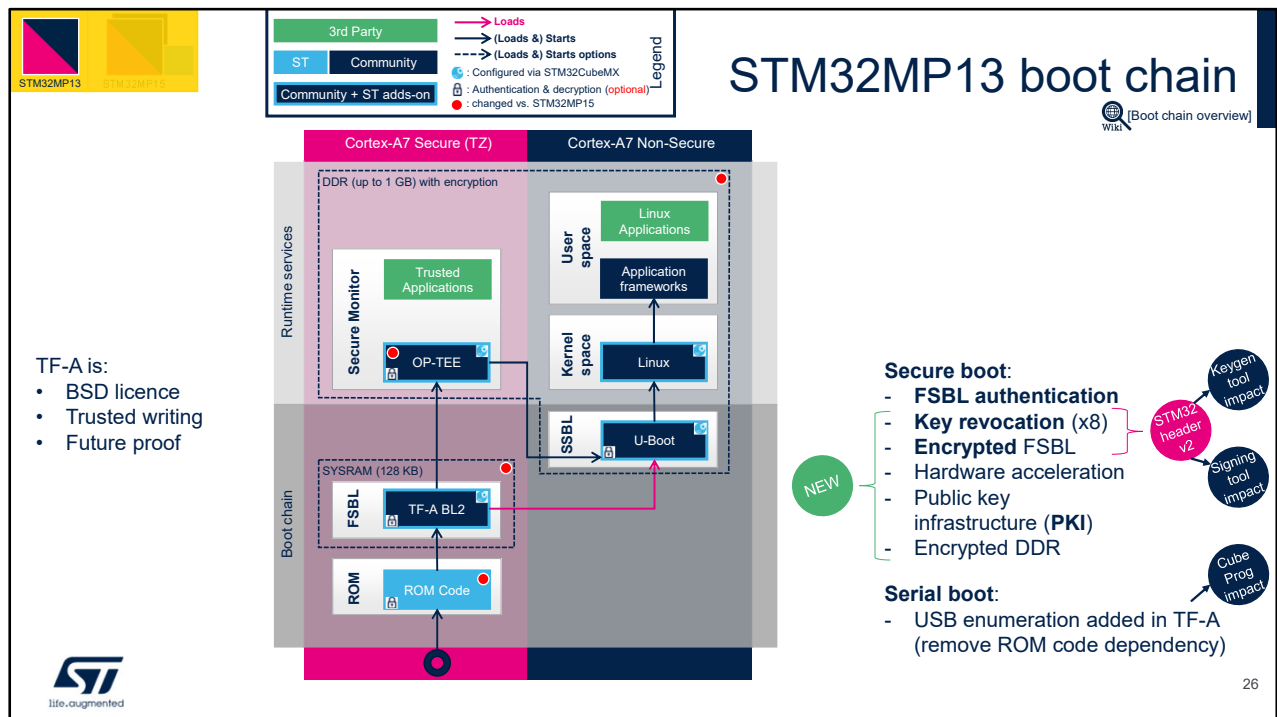
Also, the boot chain uses U-Boot as SSBL, which is covered by a GPL v2 license.

Note that the authentication is optional with this boot chain so it can run on any STM32MP1 security variant, with or

without the Secure boot option.

OP-TEE secure OS provides secure services to the non-secure world.

As seen earlier in the presentation, some services are already needed for the platform control, independently from the level of security required by the application.



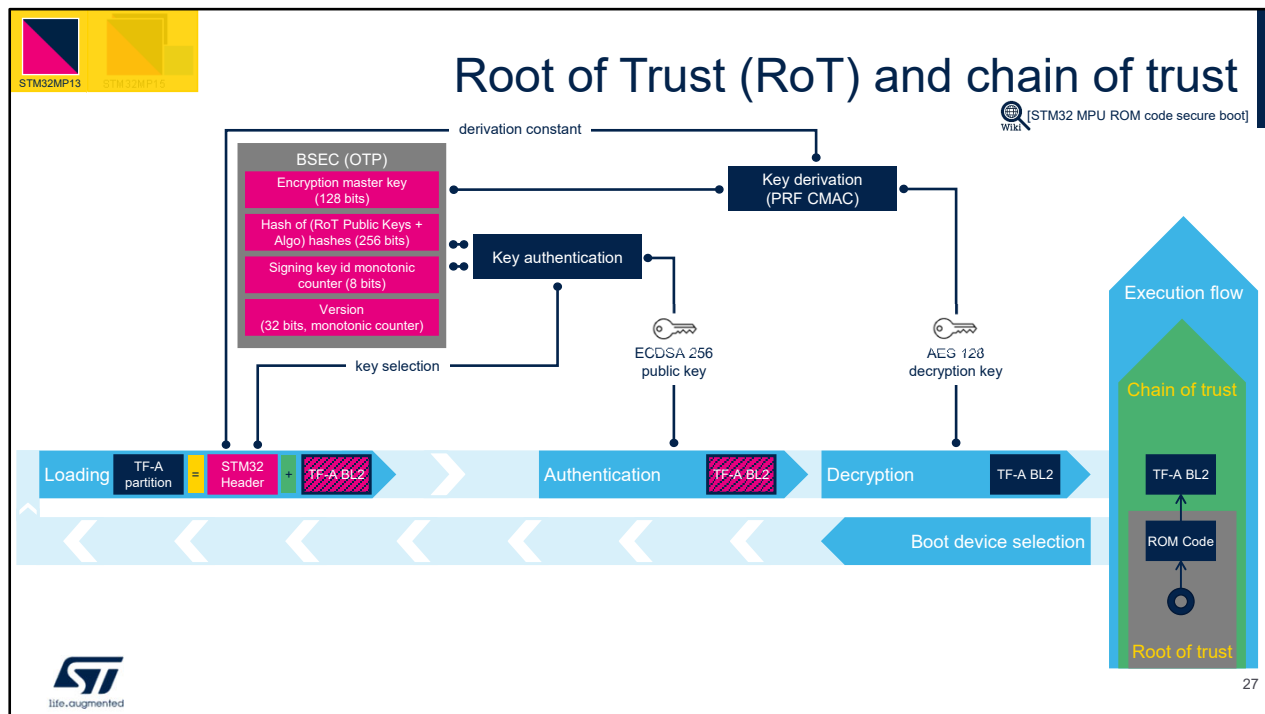
The STM32MP13 boot chain widely inherits from STM32MP15 s', apart from the Cortex-M4 context and with different memory mapping, since OP-TEE is now running in the DDR.

On the functional side, there are various improvements, listed on the right of the slide.

The most important changes, in the ROM code, are the fact that the Secure boot is now enriched with a key revocation mechanism and FSBL decryption.

All the corresponding cryptographic operations rely on hardware accelerators, with side channel attack protection. These changes have led to the definition of a new STM32 header version, implying various tool updates, listed on the right.

The public key infrastructure is enabled thanks to the use of the firmware image package in TF-A.



The chain of trust reflects the fact that each software component running on the platform is trusted by the earlier component that loaded and started it.

The chain of trust is attached to a pillar which is called the root of trust, often shortened to RoT in the technical literature.

On the STM32MP1 series, the root of trust is realized via the ROM code, which is part of the digital circuit, and various information safely provisioned in the on-time-programmable memory of the circuit during production. Once the provisioning is done, the device is put in closed state and this enables the secure boot, supported by the ROM code.

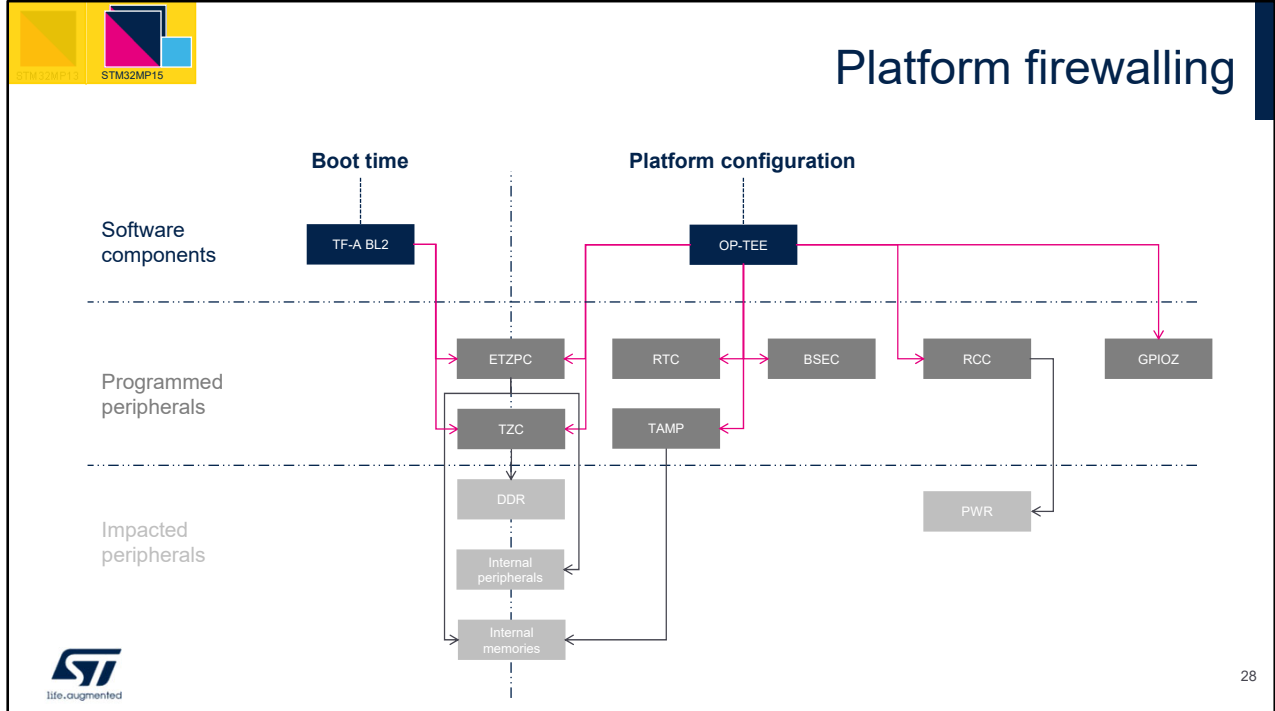
As seen earlier in this presentation, on startup, the ROM code loads the TF-A partition from the selected boot device,

which can be a flash memory or serial link.

This partition contains a header which carries various information that will be used to execute the secure boot:

- First, it contains the public key to be used to authenticate the TF-A BL2 binary. But prior to that, this key is itself authenticated thanks to information stored in OTP.
- Second, it carries the derivation constant that must be combined with the encryption master key from OTP to derive the AES key used for TF-A BL2 decryption.

Once those two steps are successfully completed, the authenticated and decrypted TF-A BL2 can safely be executed, extending the chain of trust to a new level.



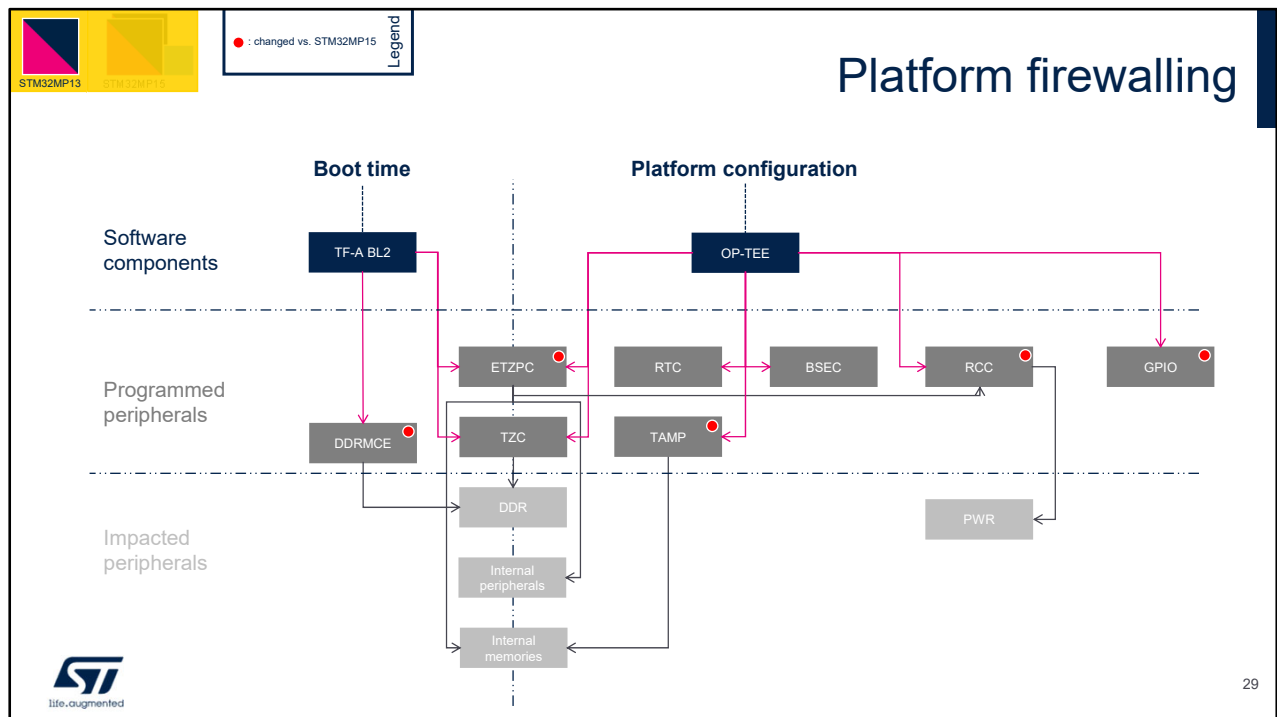
Platform firewalling corresponds to peripheral and system resource assignments to the hardware execution contexts, i.e. the Cortex-A7 secure, the Cortex-A7 non-secure and the Cortex-M4 on STM32MP15.

This configuration is done in two stages : in TF-A BL2 at boot time, then in OP-TEE to finalize the platform configuration for the rest of the execution.

Let's have a look at the programmed peripherals and impacted peripherals:

- ETZPC is a major contributor in this configuration because this is where all programmable internal peripherals and memories can be assigned to the available contexts
- TZC allows the DDR area to be split into several regions, with non-secure master address ID filtering capabilities

- RTC function control can independently be assigned to the secure or non-secure context
- TAMP peripheral owns backup registers and all the logic to be able to clear secrets on tamper event detection
- BSEC controls access to the STM32 MPU internal OTP area
- RCC can be programmed to restrict reset, clock and power control to the secure world
- Finally, the GPIOZ pins can individually be set secure or not



The firewalling programming dynamic is the same on STM32MP13 and the changes compared to STM32MP15 are highlighted with red bullets in the diagram:

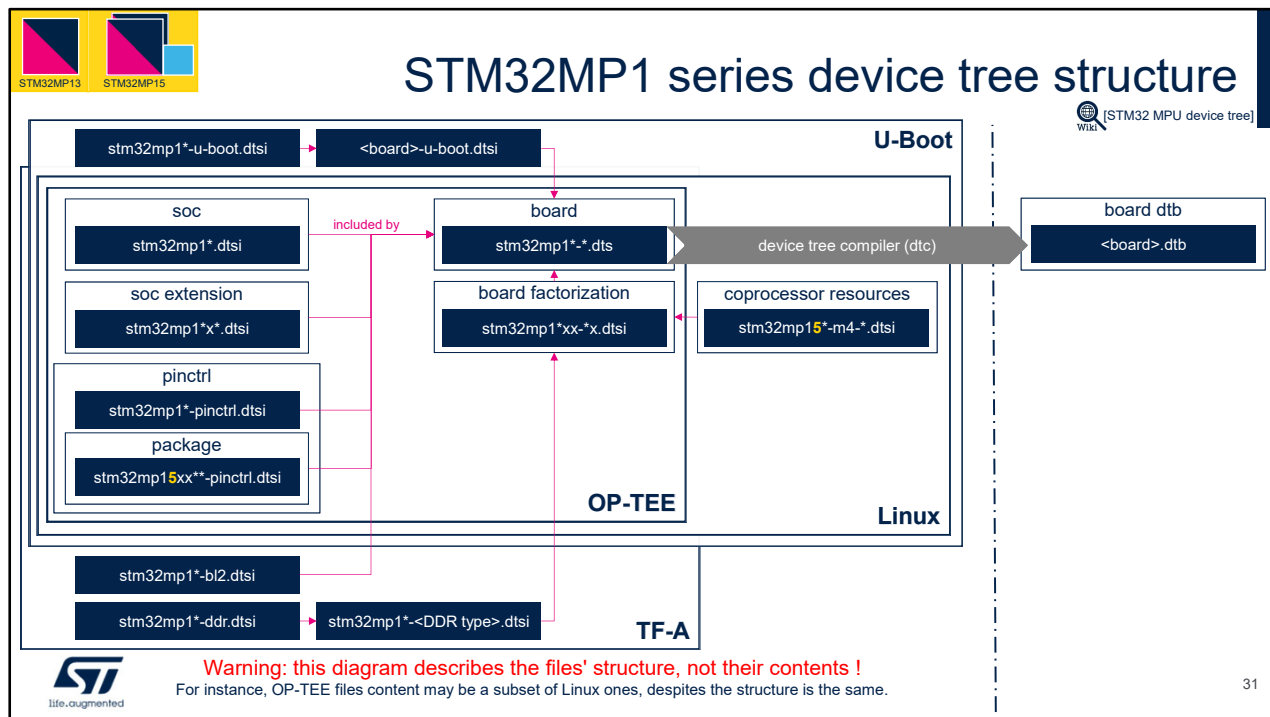
- DDRMCE is a new peripheral which allows one region to be encrypted in the DDR area. This region will typically be a subset of the whole DDR, used by the secure world firmware, i.e. OP-TEE.
- ETZPC controls the assignment for more peripherals on STM32MP13, giving better security flexibility to match various products needs
- TAMP has been enriched with new detection mechanisms and reactions
- RCC security control has been increased on two sides:
 - First, there is a better granularity for all RCC internal functions
 - Second, RCC gets ETZPC configuration directly via

dedicated connections: this allows reset and clock gating management to inherit the same assignment as the peripheral they control

- The last improvement concerns the GPIO since each pin of each bank is now individually securable.

STM32MP1 series device tree





Device tree files are used to configure all OpenSTLinux components: TF-A, OP-TEE, U-Boot and Linux kernel. This diagram shows the device tree structure in OpenSTLinux: the structure is very similar across all components, but the file contents may differ slightly. For instance, OP-TEE files may be a subset of Linux files, even though the structure is the same. Note that the DDR configuration is done by TF-A thanks to parameters embedded in dedicated device tree files. The platform firewalling and clock tree configurations are done in two steps, in TF-A, then in OP-TEE.

STM32MP1 series flash mapping

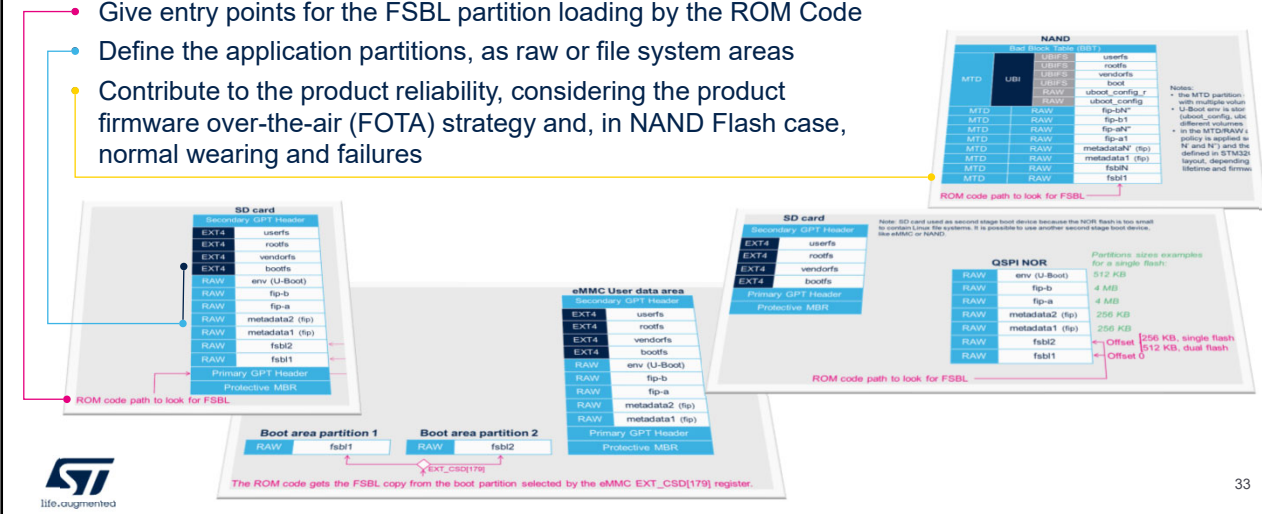




STM32MP1 series Flash mapping

[STM32 MPU Flash mapping]

- The Flash mapping must fulfil several requirements:
 - Give entry points for the FSBL partition loading by the ROM Code
 - Define the application partitions, as raw or file system areas
 - Contribute to the product reliability, considering the product firmware over-the-air (FOTA) strategy and, in NAND Flash case, normal wearing and failures



The Wiki article mentioned here will give you all the information needed to understand the default flash mapping used by STM32 MPU Embedded Software distribution, and everything you need to know to tune it for your product.

Flash mapping must be carefully prepared in order to give a bootable image to the ROM Code, and provide the various partitions expected by the application.

Beyond these pure functional aspects, it is also important to have in mind that flash mapping contributes to the product reliability, allowing firmware over-the-air updates and giving the application the means to properly manage NAND Flash failures.

Thank you

© STMicroelectronics - All rights reserved.

ST logo is a trademark or a registered trademark of STMicroelectronics International NV or its affiliates in the EU and/or other countries.

For additional information about ST trademarks, please refer to www.st.com/trademarks.

All other product or service names are the property of their respective owners.



Thank you for following this online training !

If you want to go further, with less power, you can now continue following STM32MP1 series low power management online training.