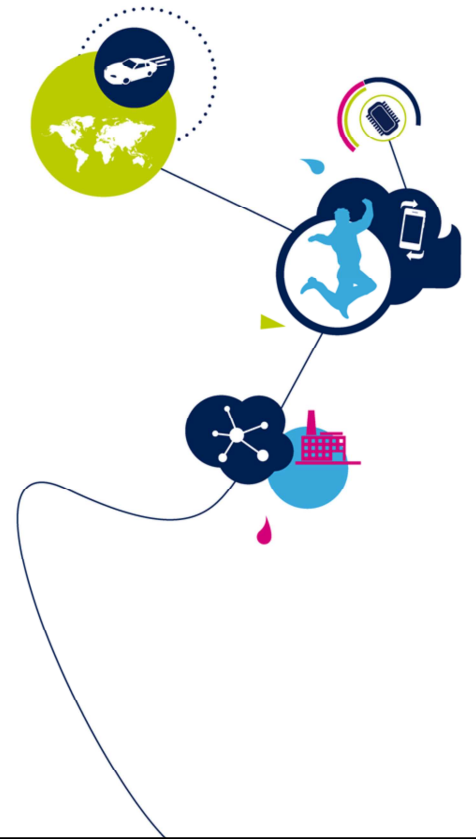# STM32MP1 SWARCH

Embedded Software architecture

Revision 1.0

Welcome to this training describing the embedded software architecture and the software distribution primarily targeting the STM32MP1 microprocessor series.

- Peripherals overview and sharing

- Software memory mapping

- STM32MPU Embedded Software

As an introduction, we are going to introduce all the internal peripherals embedded in STM32MP1 and explain their sharing capabilities across the different hardware execution contexts available on this multiple core architecture.

Then, we will spend some time on STM32MP1 memory mapping sharing capabilities. We only cover the RAM memory in this presentation since the Flash memory mapping is already covered in the Platform boot training.

Those hardware considerations will finally lead us to introduce the software components that are running on top the Arm cores, creating the STM32MPU Embedded Software distribution.
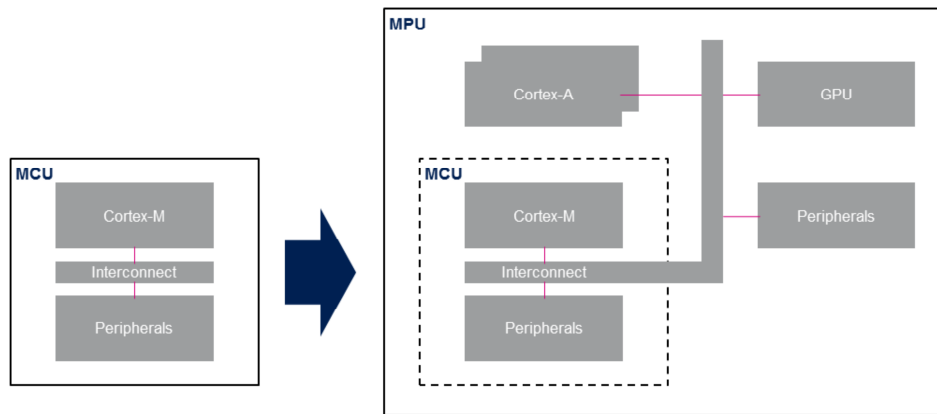
Source: ST Wiki article [Getting started with STM32 MPU devices]

Microcontroller Units, so called MCU, are built around MMU-less cores such as Arm Cortex-M, that are very efficient for deterministic operations in a bare metal or Real Time Operating System context. STM32 MCUs embed enough Static RAM and Flash memory for many applications, and can be completed with external memories.

Microprocessor Units, so called MPU, rely on cores such as Arm Cortex-A, with a Memory Management Unit to manage Virtual Memory spaces, opening the door to an efficient support of rich Operating System like Linux. A fast interconnect makes the bridge between the processing unit, high bandwidth peripherals, external memories and an optional Graphical Processing Unit (GPU).

This first MPU platform, referenced as STM32MP1, is built around a dual core Arm Cortex-A7 and an Arm Cortex-M4 cores. This platform aims at addressing multiple market

segments such as industrial, consumer, healthcare, home and building automation.

STMicroelectronics approach for a smooth transition to the MPU world consists of putting both worlds in a device, seeing the MCU as a subsystem of the MPU.

Multiple-core architecture concepts

- Hardware execution context
  - « a core and a security mode »

- Firmwares executed runtime contexts
  - 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

Runtime

Cortex-A secure context

Cortex-A non secure context

Cortex-M (non secure) context

Boot chain

Boot time
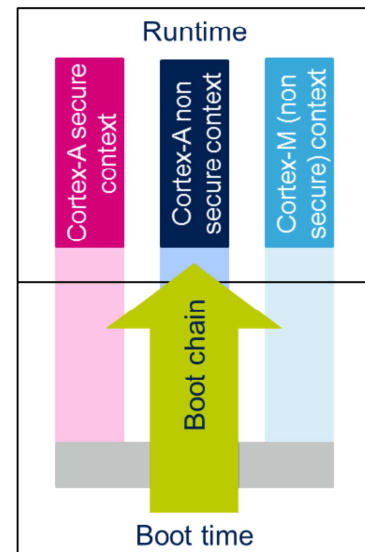
Source: ST Wiki article [Getting started with STM32 MPU devices]

Some new concepts need to be introduced for a good understanding of a multiple-core architecture such as the STM32MP1 one.

First of all, an hardware execution context is defined by a core and a security mode.
On the STM32MP1 microprocessor, there are three hardware execution contexts:
- Arm Cortex-A secure, also called Trustzone
- Arm Cortex-A non secure
- Arm Cortex-M, supporting only the non-secure mode

Each hardware execution context may support a part of the boot chain execution at boot time, and supports the execution of a firmware at runtime. On STM32MP1 microprocessor, there are also three runtime contexts:
- Arm Cortex-A secure executes OP-TEE secure OS
- Arm Cortex-A non secure executes Linux OS

- Arm Cortex-M executes STM32Cube

The term "peripheral assignment" is used to identify the action of assigning a set of peripherals to a runtime context. Each peripheral can be assigned to a runtime context or shared across several contexts.
Shared peripherals are typically system resources, like RCC for the reset and clock control.

Peripherals sharing — 5

Source: ST Wiki article [STM32MP1 peripherals overview]
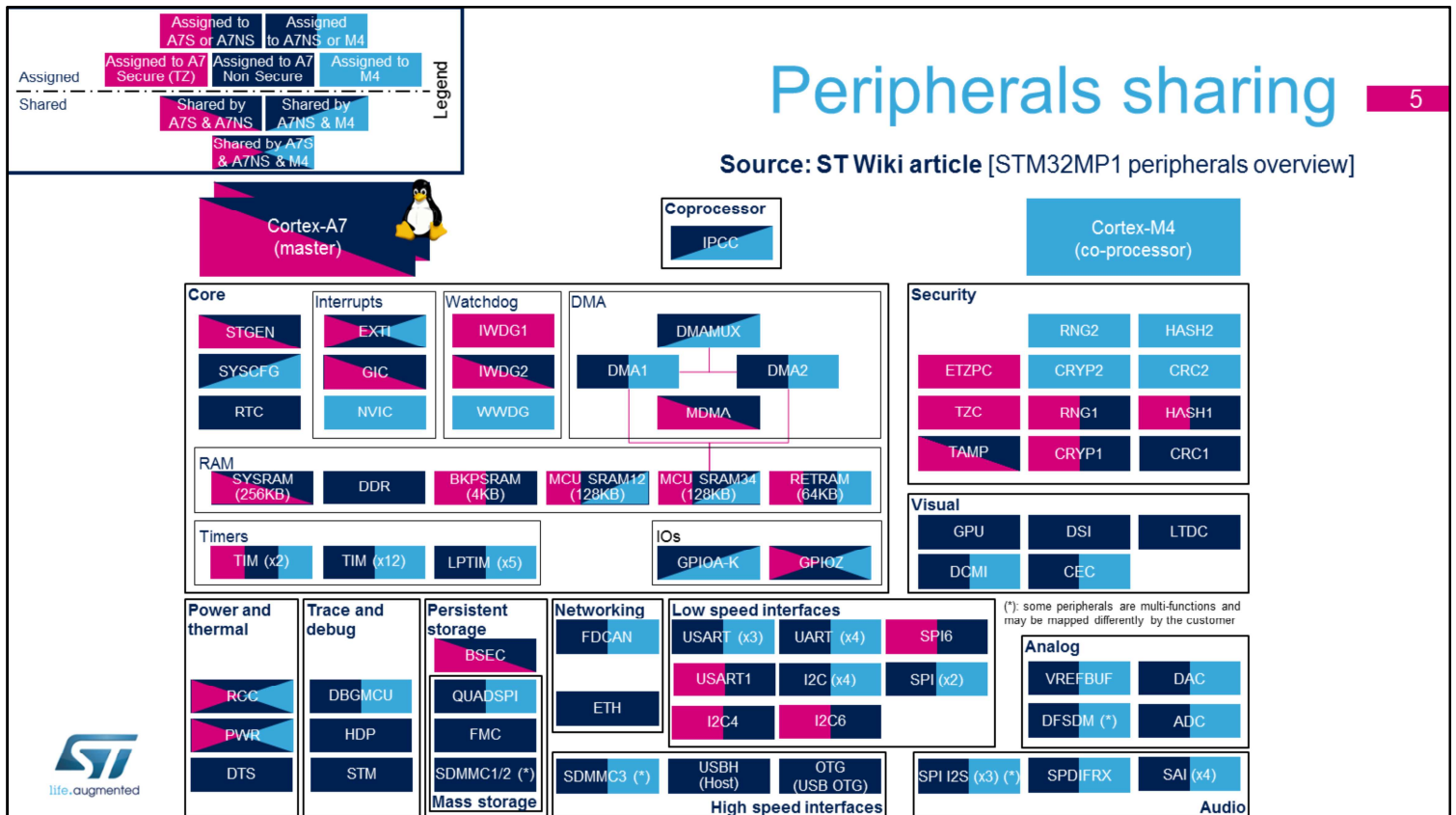
In line with the concepts introduced on the previous slide, the legend of this diagram distinguishes two possible states for each peripheral:
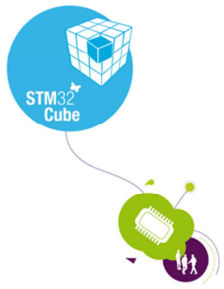
- "Assigned" means that only one hardware execution context is using the peripheral at runtime. A single colour box means that the assignment is static whereas a double or triple colour box with vertical separator means that a user choice is needed to assign the peripheral to a given context, depending on its application needs. This assignment can either be done with STM32CubeMX tool or manually, 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 ensures there is no contention between the given contexts when they access to 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 ST Wiki, starting from the [STM32MP15 peripherals overview] article.

Notice that this diagram shows STMicroelectronics recommendations or choices of assignment in STM32 MPU Embedded Software distribution. Additional possibilities might be described in STM32MP15 reference manual and may be considered later on in our distribution.

	Boot ROM	Boot loader	Cortex-A7 secure	A7NS	Cortex-M4
✓ USART1		☐	☑	☐	
✓ USART2	☐	☐		☑	☐
✓ USART3	☐	☐		☐	☑
✓ USART6	☑	☑		☑	☐

This screenshot shows an example of assignment as it can be done with STM32CubeMX:
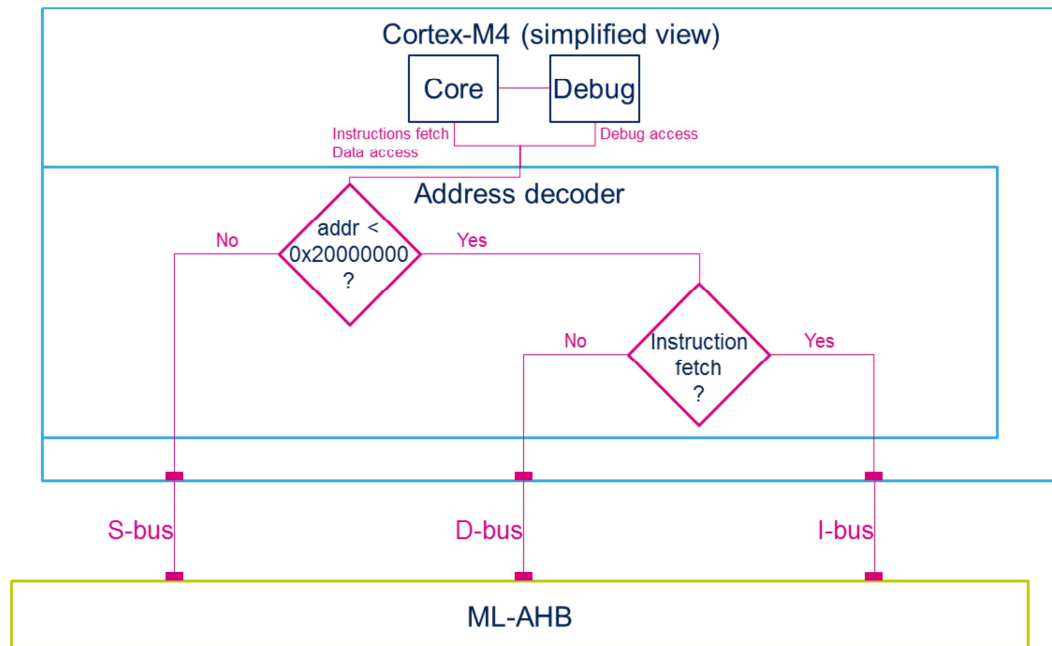
- USART1 is assigned to the Cortex-A7 secure for OP-TEE
- USART2 is assigned to the Cortex-A7 non secure for Linux
- USART3 is assigned to the Cortex-M4 for STM32Cube

Note the two columns that were never mentioned up to now, on the left, used for USART6:

- the « Boot ROM » column allows the ROM code to be able to boot from this instance. This choice limits the pins possibilities to the ones supported by the ROM code.
- the « Boot loader » column gives visibility on the selected peripheral for the boot loader. The main target of this choice is to limit the size of the device tree generated for the boot loader.

The ROM code and the boot loader are presented in the

Platform boot training.

Cortex-M4 ports

Cortex-M4 (simplified view)

Core — Debug

Instructions fetch / Data access — Debug access

Address decoder

addr < 0x20000000 ?
No / Yes

Instruction fetch ?
No / Yes

S-bus   D-bus   I-bus

ML-AHB

Source: ST Wiki article [STM32MP1 RAM mapping]

This slide focuses on a Cortex-M4 specificity, since it is connected to the MLAHB interconnect via three different ports, listed below and shown in the figure:
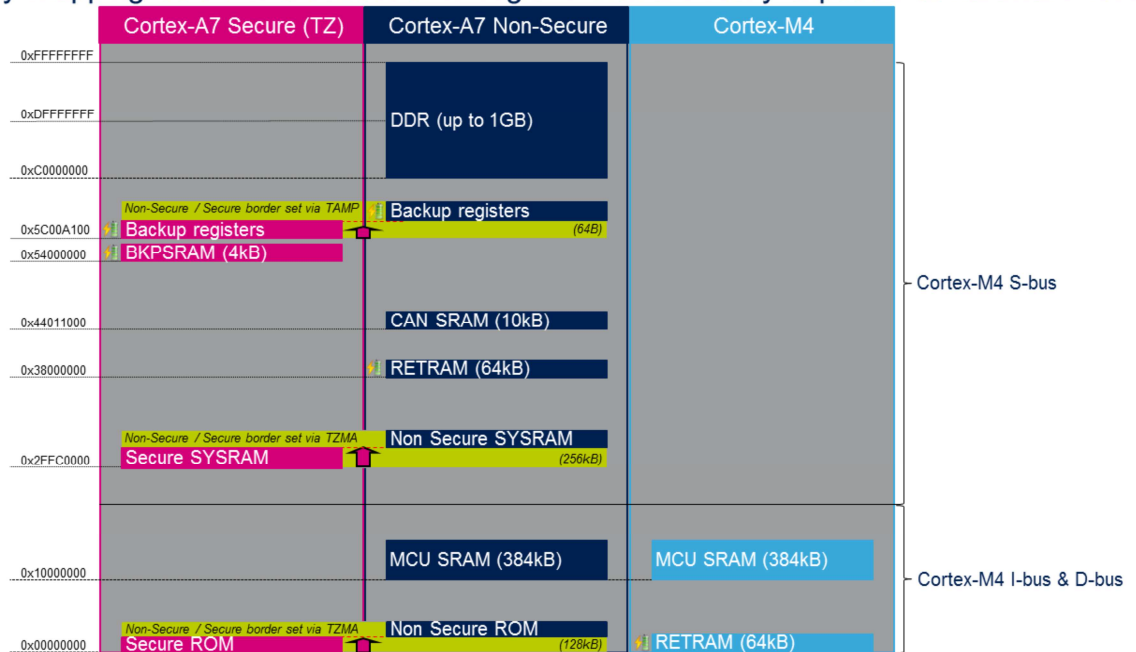- I-bus is used to fetch code instructions in the 0x00000000--0x1FFFFFFF address range.
- D-bus is used to read/write data in the 0x00000000--0x1FFFFFFF address range.
- S-bus is used for all accesses in the 0x20000000--0xFFFFFFFF address range. All STM32MP15 internal peripherals registers are mapped in this range.

Balancing the Cortex-M4 firmware accesses among those ports can allow the tuning of the system performances. This is why the MCU SRAM is defined in the first address range, from 0x10000000, but it is also visible in the second range, from 0x30000000. For the sake of simplicity, this second range is not shown in the next slides but keep in mind that it exists.

Legend
⚡: memory content preserved during standby

Software memory mapping

- The memory mapping below is a subset of all regions that are really exposed at hardware level.

	Cortex-A7 Secure (TZ)	Cortex-A7 Non-Secure	Cortex-M4

0xFFFFFFFF

0xDFFFFFFF — DDR (up to 1GB)

0xC0000000

Non-Secure / Secure border set via TAMP
0x5C00A100 — Backup registers | Backup registers (64B)
0x54000000 — BKPSRAM (4kB)

0x44011000 — CAN SRAM (10kB)

0x38000000 — RETRAM (64kB)

Non-Secure / Secure border set via TZMA
0x2FFC0000 — Secure SYSRAM | Non Secure SYSRAM (256kB)

0x10000000 — MCU SRAM (384kB) | MCU SRAM (384kB)

Non-Secure / Secure border set via TZMA
0x00000000 — Secure ROM | Non Secure ROM (128kB) | RETRAM (64kB)

Cortex-M4 S-bus
Cortex-M4 I-bus & D-bus

life.augmented

This diagram maps the various RAM and ROM regions on the different execution contexts that use them in STM32 MPU Embedded Distribution.

It is important to notice that the RETRAM appears twice here:

- one time on Cortex-A7 non secure side for coprocessor firmware loading
- one time on Cortex-M4 side for coprocessor firmware execution, that starts to boot from the 0x00000000
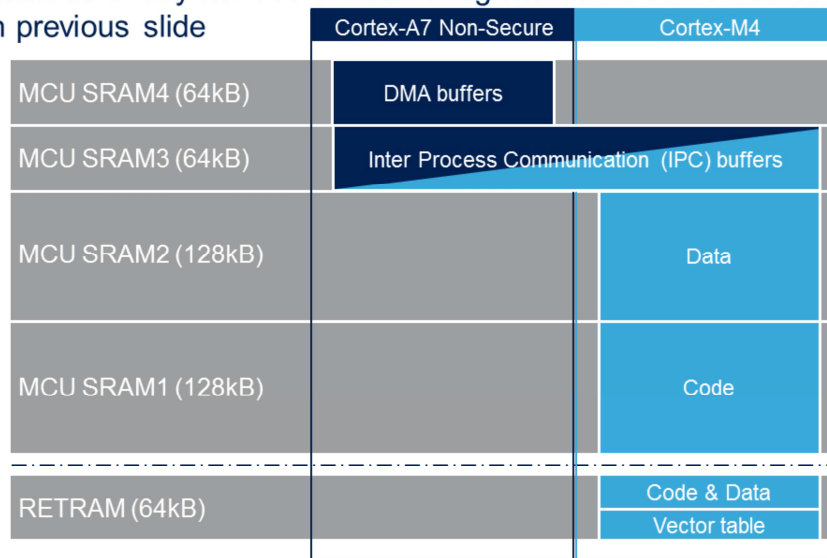- address

This diagram is available on the same ST Wiki article as the previous one: [STM32MP1 RAM mapping].
The Wiki also gives information on the way each one of those regions is used.

Shared RAM memory mapping

- Notice that each core may not see the same regions at the same address, as already explained on previous slide

	Cortex-A7 Non-Secure	Cortex-M4
MCU SRAM4 (64kB)	DMA buffers	
MCU SRAM3 (64kB)	Inter Process Communication (IPC) buffers	
MCU SRAM2 (128kB)		Data
MCU SRAM1 (128kB)		Code
RETRAM (64kB)		Code & Data / Vector table

- Each customer can of course tune this mapping (regions location and sizes) to fit with his product needs

This slide shows the RAM banks available on STM32MP1 microprocessors and the typical mapping applied to STM32 MPU Embedded Software distribution:

- RETRAM is used by the Cortex-M4 to put its vector table, plus some code and data
- MCU SRAM1 and SRAM2 can be used to put the remaining Cortex-M4 firmware code and data
- MCU SRAM3 is typically used to map the inter process communication buffers, that are further described in the coprocessor management training
- MCU SRAM4 can be used to put DMA buffers for the Cortex-A7 when high bandwidth is needed when using DMA1 or DMA2 instances

It is not mandatory to align this mapping on regions borders but this can represent an interest since hardware isolation for Cortex-M4 memories is supported with a per-bank granularity.

Source: ST Wiki article [STM32MPU Embedded Software architecture overview]
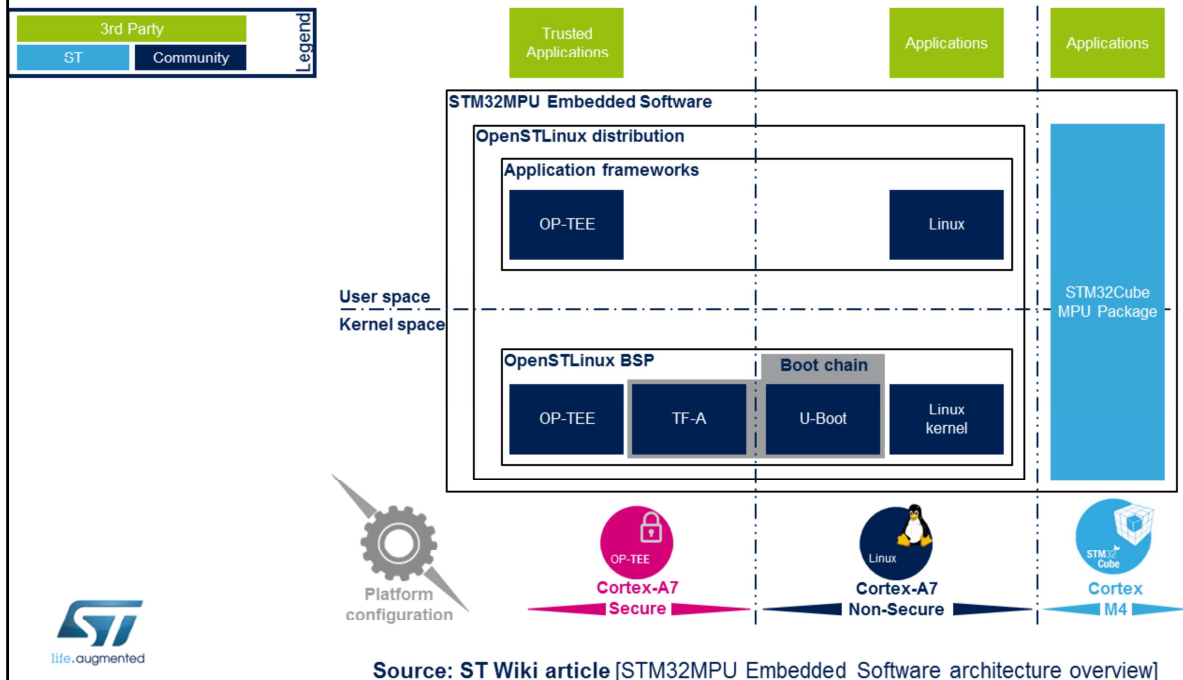
Let's now introduce the STM32MPU Embedded Software distribution.

This diagram explains the terminology used to define the main components of the STM32MPU Embedded Software. The OpenSTLinux distribution, running on the Arm® Cortex®-A, includes:

- The OpenSTLinux BSP with:
    - The boot chain based on TF-A and U-Boot.
    - The OP-TEE secure OS running on the Arm® Cortex®-A in secure mode.
    - The Linux® kernel running on the Arm® Cortex®-A in non-secure mode.
- The application frameworks are the middleware relying on the BSP and providing APIs:
    - On the OP-TEE side to run Trusted Applications that allow to manipulate secrets, not visible from the Linux and STM32Cube MPU Package
    - On the Linux side to run Applications that typically

interact with the user via a display, a touchscreen, etc.

The STM32Cube MPU Package is running on the Arm® Cortex®-M: it is based on HAL drivers and the middleware, like other STM32 microcontrollers, completed with coprocessor management.

This diagram is available as a clickable image in the ST Wiki, allowing to easily jump into the area to explore and get more details about it.
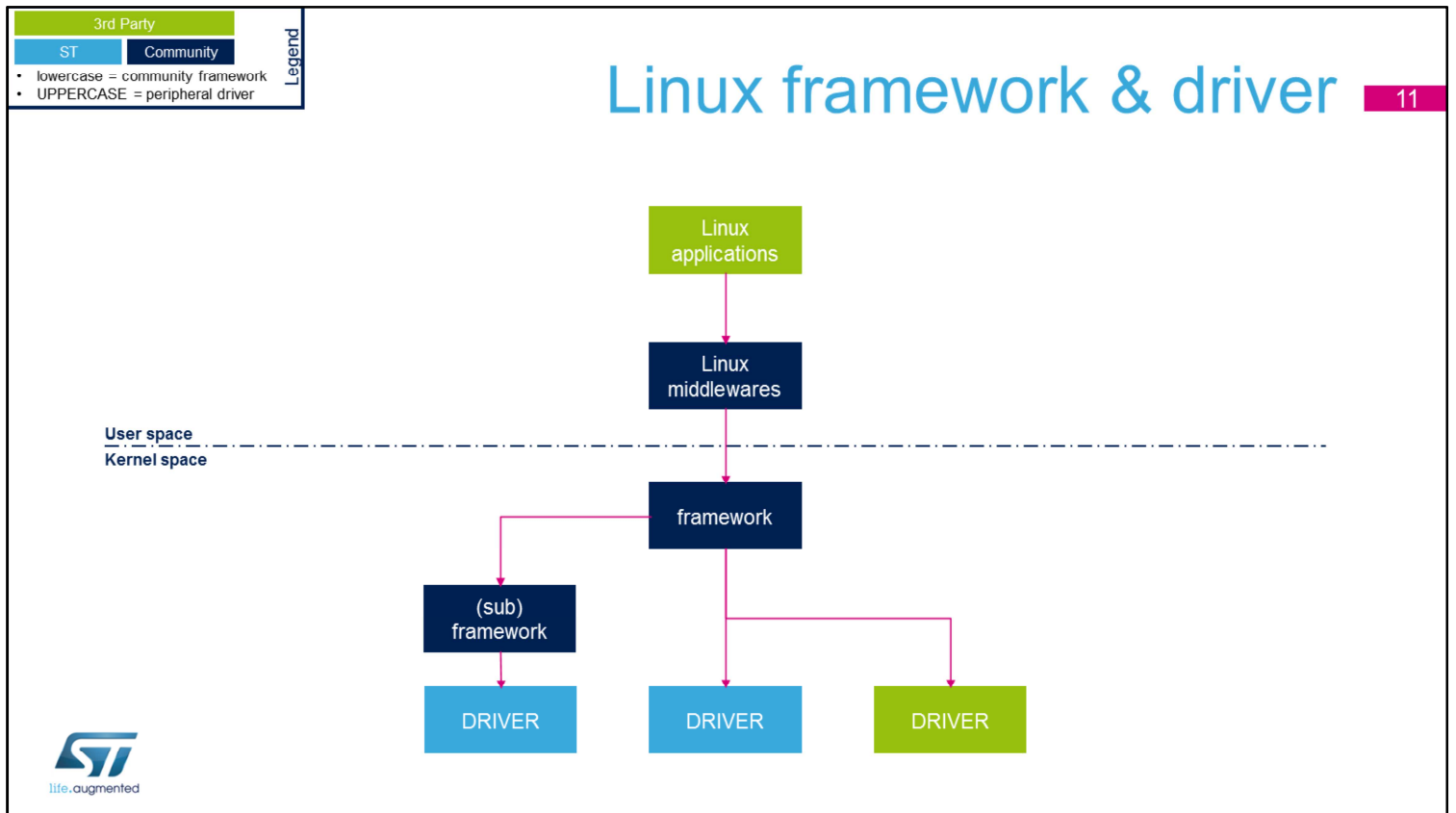
This slide aims at giving a very high level overview on how a Linux system is organized, in order to ease understanding of the content of the next slide.

On the user side, we have applications that usually rely on middleware to invoke the Linux kernel.

Linux drivers are not directly exposing their API to the user space. Instead of that, they are registered to frameworks that themselves expose unified APIs to the user space. The framework concept is very important because it ensures that an application working on a stable framework API will also work on a future Linux kernel version and that it can as well work on other platforms implementing other drivers.

On the left, you can see that some frameworks may even have some sub-frameworks. This is for instance the case with SD card support, that is a sub part of the MMC
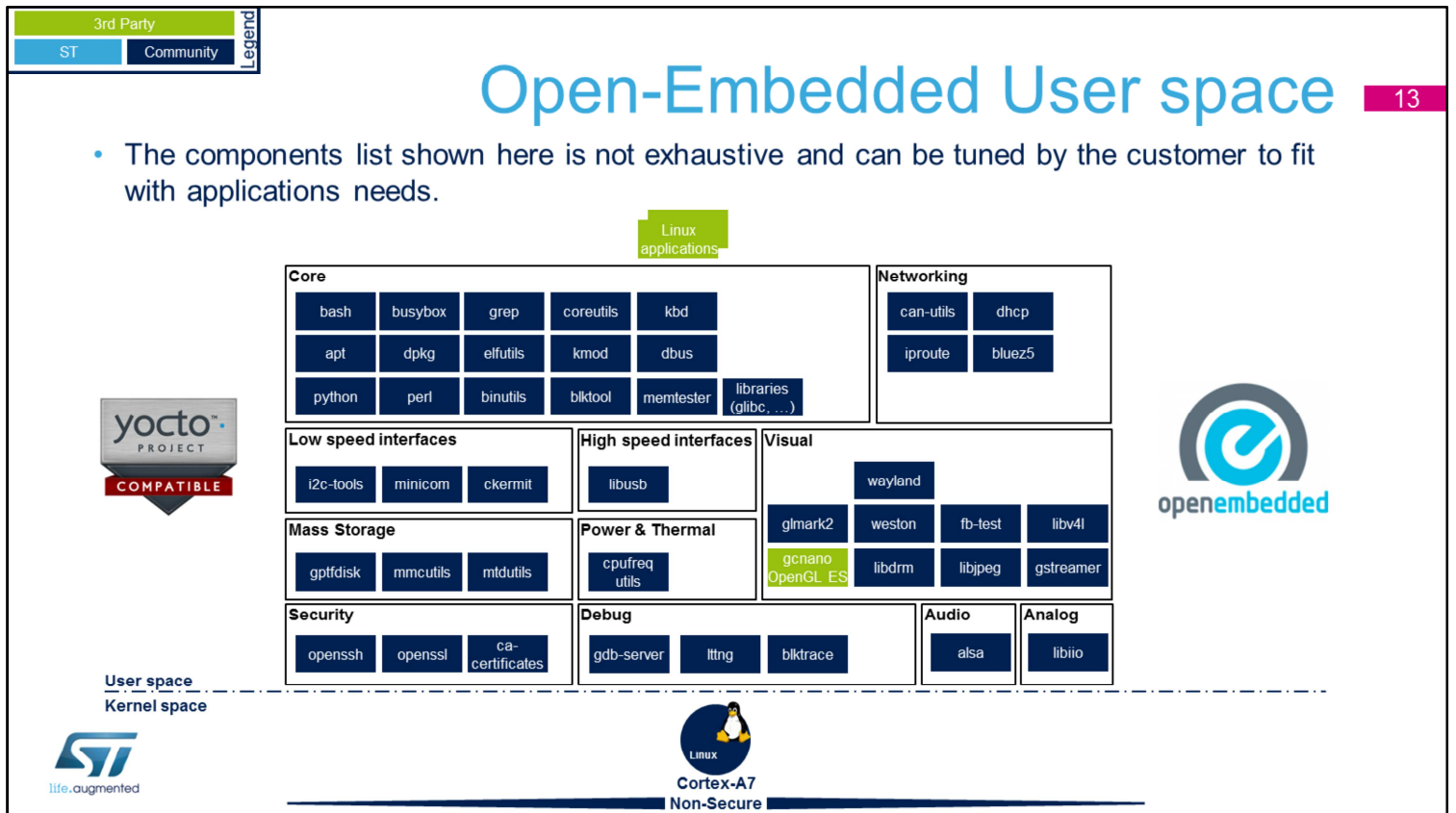
framework.
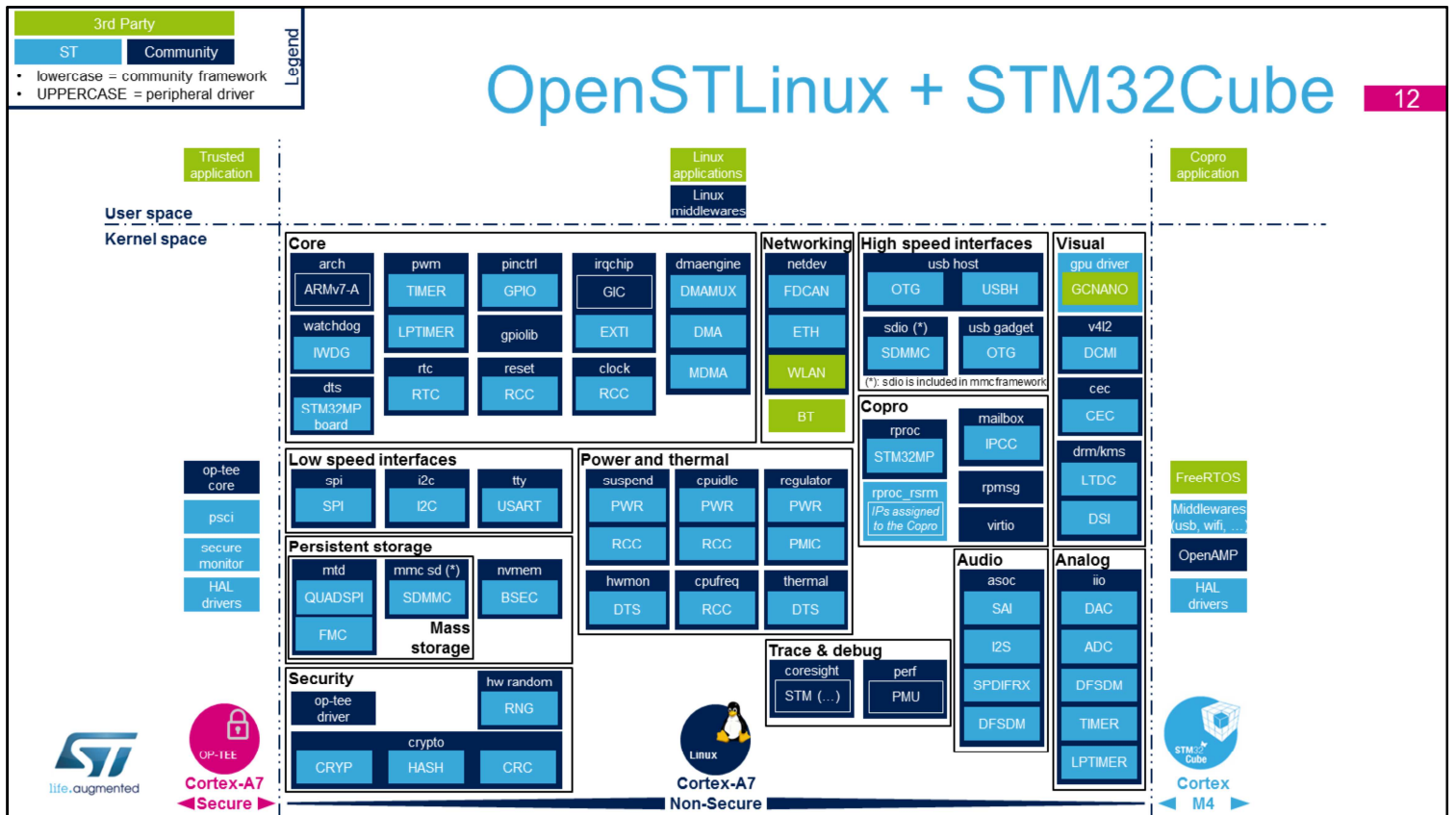
This slide shows the STM32MP1 peripheral drivers and their respective Linux frameworks:

- The frameworks names are written in lowercase
- The peripheral driver names are written in uppercase
- The color code indicates the origin of the source code for each component: third party, community or STMicroelectronics

This view is also available in the ST Wiki.

The STM32MPU Embedded Software distribution is build thanks to the Open-Embedded build framework for embedded Linux platforms.

This diagram shows the usual components that are embedded in our distribution. Feel free to explore the ST Wiki and Open-Embedded online distribution to customize this setup for your needs.