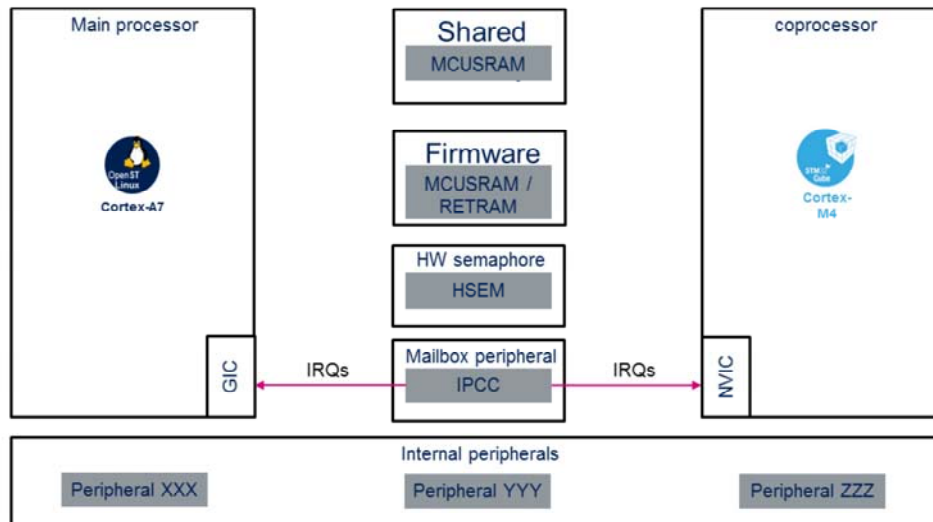


STM32MP1 - COPROC

Coprocessor management
Revision 1.0



Hello, and welcome to this presentation of the STM32MP1 coprocessor management. It describes how the processors interact through commands and events, exchange data and share resources.



The STM32MP1 multiprocessor system allows independent firmware to run on two CPU cores.

- The Master Arm Cortex-A7 processor is optimized to run Linux based OS.
- The Slave (or coprocessor) Arm Cortex-M4 processor can run RTOS optimized for micro-controllers or a bare-metal application.

In addition, several integrated STM32 peripherals can be assigned to one of these processors.

Then, two internal memory regions are shared between the master and the slave processors. These memories are used to load and execute the coprocessor firmware, but also to define common structures, like shared buffers for inter-processor communication.

Furthermore, for inter-processor communication, a specific peripheral named Inter Processor Communication Controller (or IPCC) enables signaling by dedicated mailboxes.

Finally a hardware semaphore (or HSEM) can be used to protect shared resources from concurrent accesses.

- Coprocessing management is a mechanism put in place to handle multicore in heterogeneous asymmetric architecture.
- In order to manage a coprocessor, a list of services are proposed:
 - Load and control Cortex®-M4 firmware, by Cortex®-A7 (Linux or Uboot).
 - Inter-processor communication based on RPMmsg messages and mailboxes.
 - Resources management:
 - Peripheral assignment to a core (with access right request)
 - System resources management (resources that are common to both cores)



Before entering the details, it is important to understand the concepts involved in the management of a coprocessor in a multi-processor system.

The first one is the load and the control of the Cortex-M4 core. The Cortex-A7 core is in charge of loading the Cortex-M4 firmware and controlling the Cortex-M4 core reset.

Then the inter-processor communication is ensured by the RPMmsg protocol that relies on a shared memory and the IPCC peripheral.

Finally a resource management service is proposed to provide facilities to:

- Assign a peripheral to a core and control the exclusive access to this peripheral by this core only,
- Manage access to the common (or system) resources shared between the two cores, for instance clocks or GPIOs.

- On Cortex-A7, Linux OS integrates frameworks that enable the control and the communication with remote processors:
 - **RemoteProc** - The generic Remote Processor framework enables the control (power on, load firmware, power off) of remote processors.
 - **RPMsg** - the Remote Processor Messaging is a VirtIO-based messaging bus that allows kernel drivers to communicate with remote processors available on the system.
 - **VirtIO** - VirtIO framework supports virtualization. It provides an efficient transport layer based on shared ring buffer (Vring).
- On Cortex-M4, **OpenAMP** is a library integrating RemoteProc and RPMsg frameworks for baremetal and Real Time OS (freeRTOS, Zephyr...).



In terms of Software, the management of a coprocessor is done through three main frameworks.

The Remote Processor or RemoteProc framework is responsible for loading the firmware and all resources required by the Cortex-M4 coprocessor to properly operate. The Remote Processor Messaging framework named RPMsg is used for the inter-processor communication and relies on the VIRTUAL Input Output or Virtio [Virt][I][O] framework.

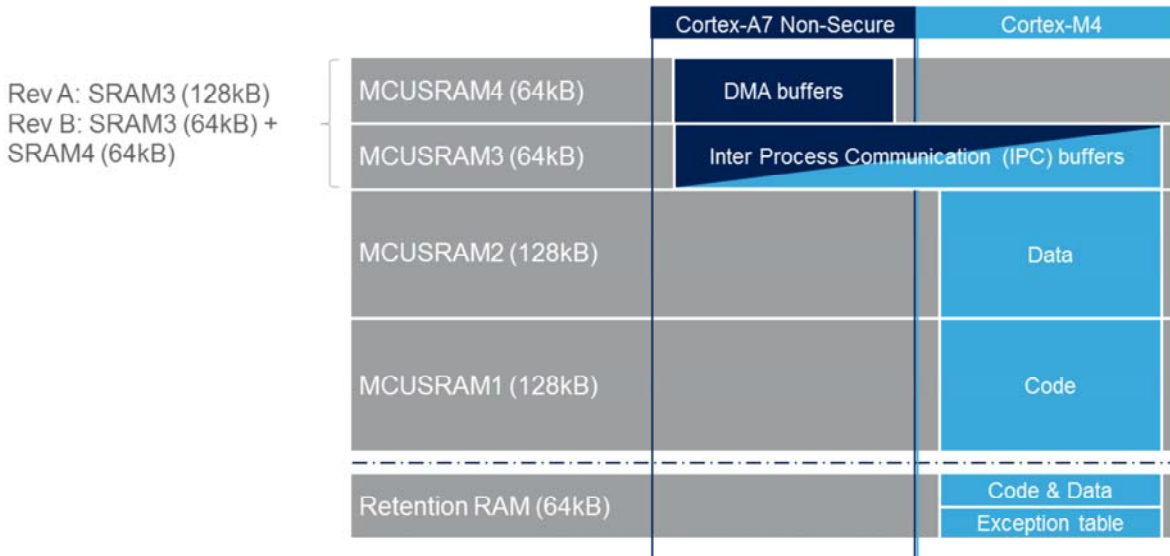
This VIRTUAL Input Output framework provides a mechanism to manage the shared ring buffer pool named Vring [V][ring]. On Cortex-A7, these frameworks are native in the Linux kernel distribution. The RemoteProc framework is also integrated in the U-boot, allowing the pre-load of the Cortex-M4 firmware before the Linux firmware.

On Cortex-M4, the frameworks are available by integrating the OpenAMP [Open][AMP] Library for bare metal and Real time OS.

Shared RAM memory mapping

5

- Cortex-M4 firmware must start at address 0 of the RETRAM, but can be partially located in MCUSRAM

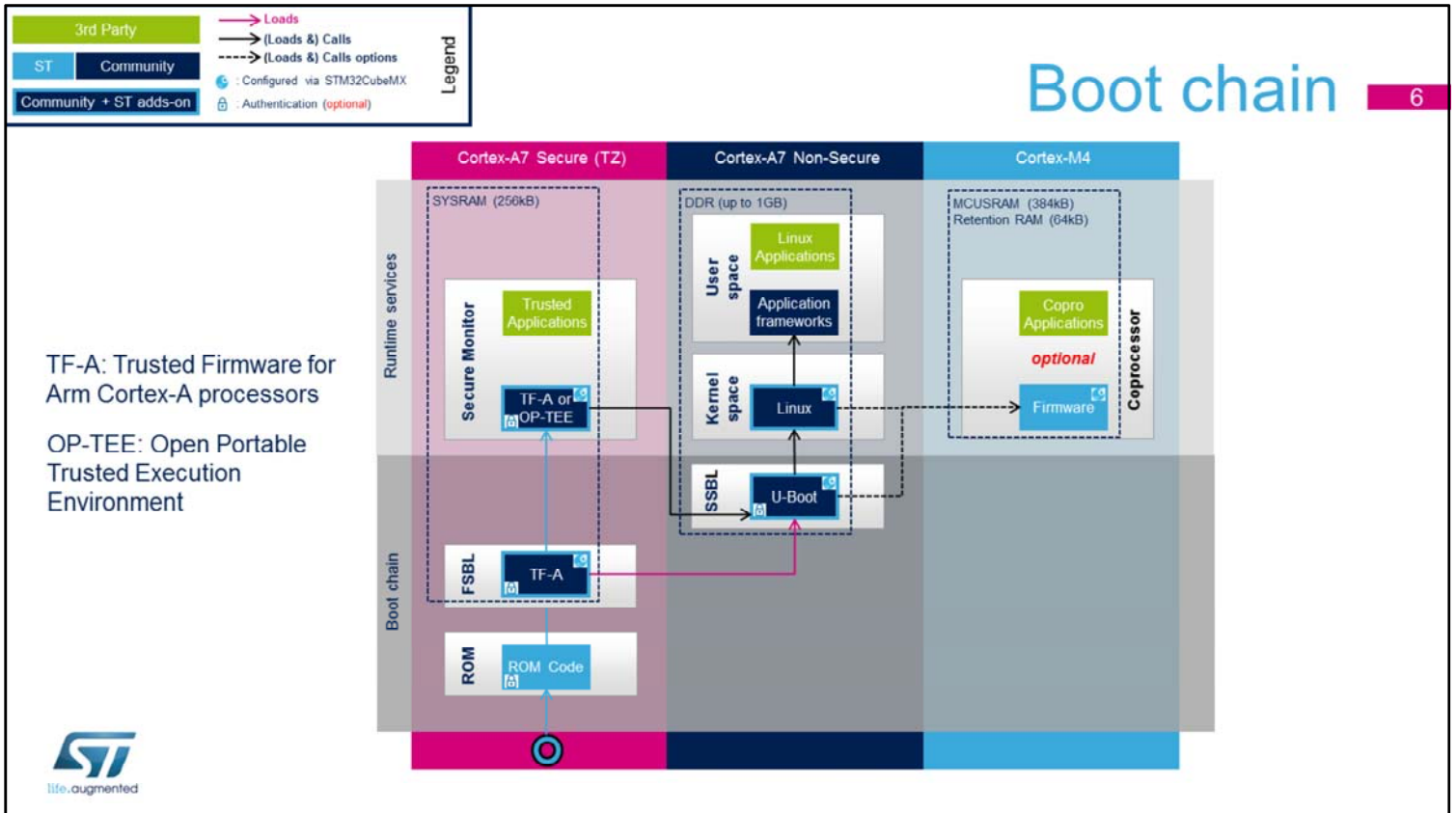


• Each customer can of course tune this mapping (regions location and sizes) to fit with his product needs

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

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



The Trusted boot chain is the default solution delivered by ST Microelectronics, with a complete feature set.

The Cortex-M4 firmware can be loaded and started by the Linux OS or by the secondary bootloader, for instance U-boot.

In «normal» mode, the firmware is stored in the file system and is loaded by the Linux user-land through the RemoteProc file system interface.

In «early boot» mode, the firmware is installed in the BootFS partition and is loaded by U-boot before the Linux firmware starts.

- Firmware is stored in an ELF format file.
- In the firmware binary, an optional resource table is defined at a specific “.resource_table” section known by both Cortexes. This table is used to define common resources:
 - Provide a user sysfs interface to access coprocessor traces for debug.
 - Load the RPMsg and VirtIO Frameworks to support messaging services.

```

/* Resource table with trace and Vring for IPC */
struct remote_resource_table __resource__attribute__((used)) rproc_resource = { \

    .version = 1, \
    .num = 2, \
    .reserved = {0, 0}, \
    .offset = { \
        offsetof(struct remote_resource_table, \
                rpsmsg_vdev), \
        offsetof(struct remote_resource_table, \
                cm_trace), \
    }, \
    /* Virtio device entry for IPC */
    .rpsmsg_vdev= { \
        RSC_VDEV, VIRTIO_ID_RPMSG, 0, \
        RPMSG_IPU_C0_FEATURES, 0, 0, 0, \
        NUM_VRINGS, {0, 0}, \
    }, \
    /* Vring rsc entry - part of vdev rsc entry */
    .rpsmsg_vring0 = {VRING_TX, VRING_ALIGN, VRING_SIZE, 1, 0}, \
    .rpsmsg_vring1 = {VRING_RX, VRING_ALIGN, VRING_SIZE, 2, 0}, \
    /* trace buffer declaration accessible from Cortex-A7*/
    .cm_trace = { \
        RSC_TRACE, \
        (uint32_t)system_log_buf, \
        sizeof(system_log_buf), 0, "cm4_log", \
    }, \
};
    
```



The Cortex-M4 firmware is stored in ELF format to be loaded by the RemoteProc framework from U-boot or Linux. In addition to the code and data sections, a specific section can be defined to support features related to the coprocessor management. This section consists of a resource table structure, which is parsed by the RemoteProc Linux framework during the firmware load phase.

The first feature declared in this table is the remote processor trace buffer that offers the possibility to output the Cortex-M4 logs to a ring buffer. This buffer can be monitored on the Cortex-A7 side. The address and the size of the ring buffer is declared in the resource table.

The second one is the inter-processor communication protocol. To enable the RPMMSG protocol, the Virtio device and associated virtio ring descriptors have to be declared in the table. The Cortex-A7 RemoteProc framework, acting as a master, is in charge of allocating the associated RPMMSG

buffers in MCUSRAM and completing this table in consequence.

The inter-processor communication

8

- By Inter-processor communication we refer to messaging service put in place to allow communication between the Cortex-A and the Cortex-M cores:
 - Message service is managed by the RPMsg and Virtio frameworks, through a shared memory
 - Doorbell/mailbox service is offered by the IPCC internal peripheral.

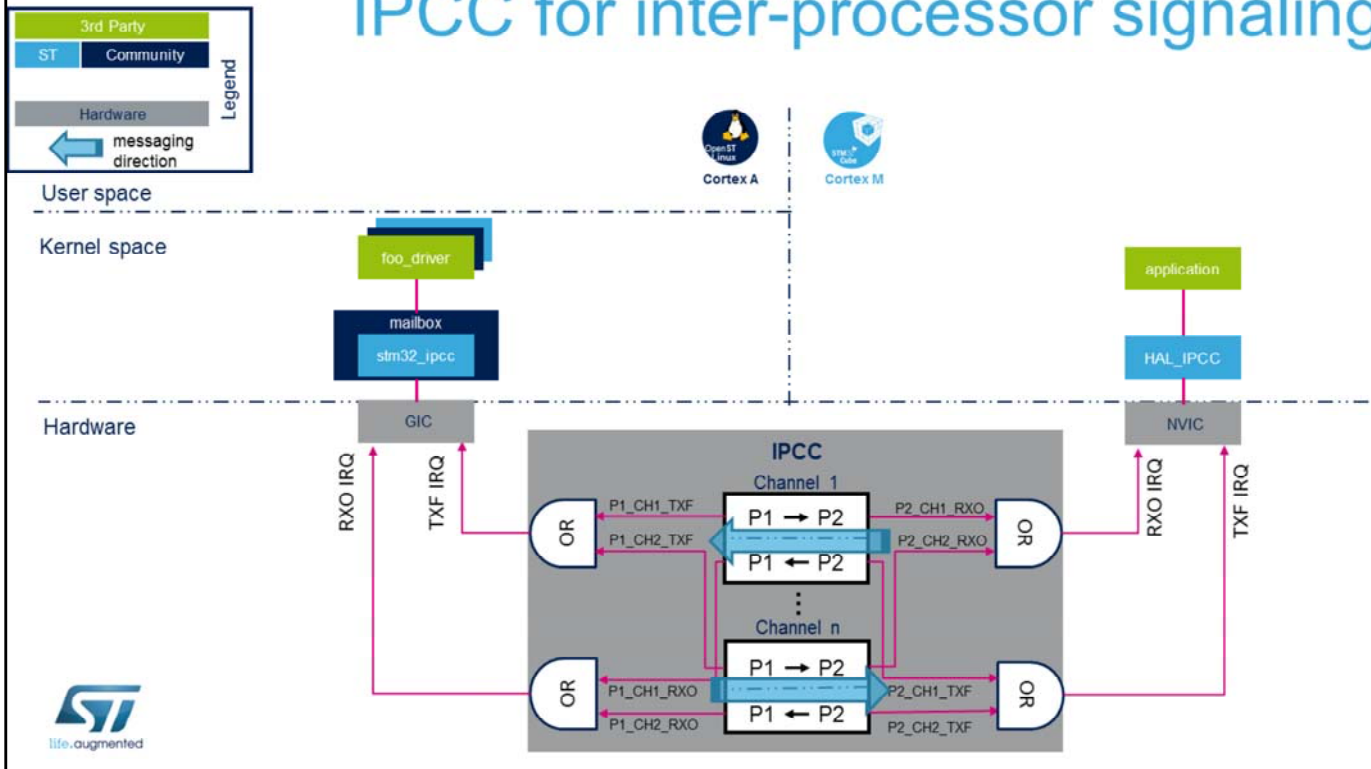


The inter-processor communication between the Cortex-M4 and the Cortex-A7 processors relies on The RPMsg messaging service.

The RPMsg and Virtio frameworks are in charge of managing the buffers involved in the communication. Then a doorbell or mailbox mechanism is in place to inform the processors that a new message is available. This signaling is generated thanks to the Inter Processor Communication Controller (IPCC).

IPCC for inter-processor signaling

9



The IPCC peripheral integrated in the STM32MP1 offers 6 bidirectional channels for the communication between the Cortex-A7 and the Cortex-M4.

Principle is as following:

- A Core, for instance the Cortex-A7, sets a channel flag which generates a Rx occupied (RXO) interruption on the other Core, here the Cortex-M4.
- The Cortex-M4 clears the flag to free the channel. This generates the TX free (TXF) interrupt on Cortex-A7.

It is important to understand that the IPCC peripheral does not manage any buffer, it works only like a doorbell.

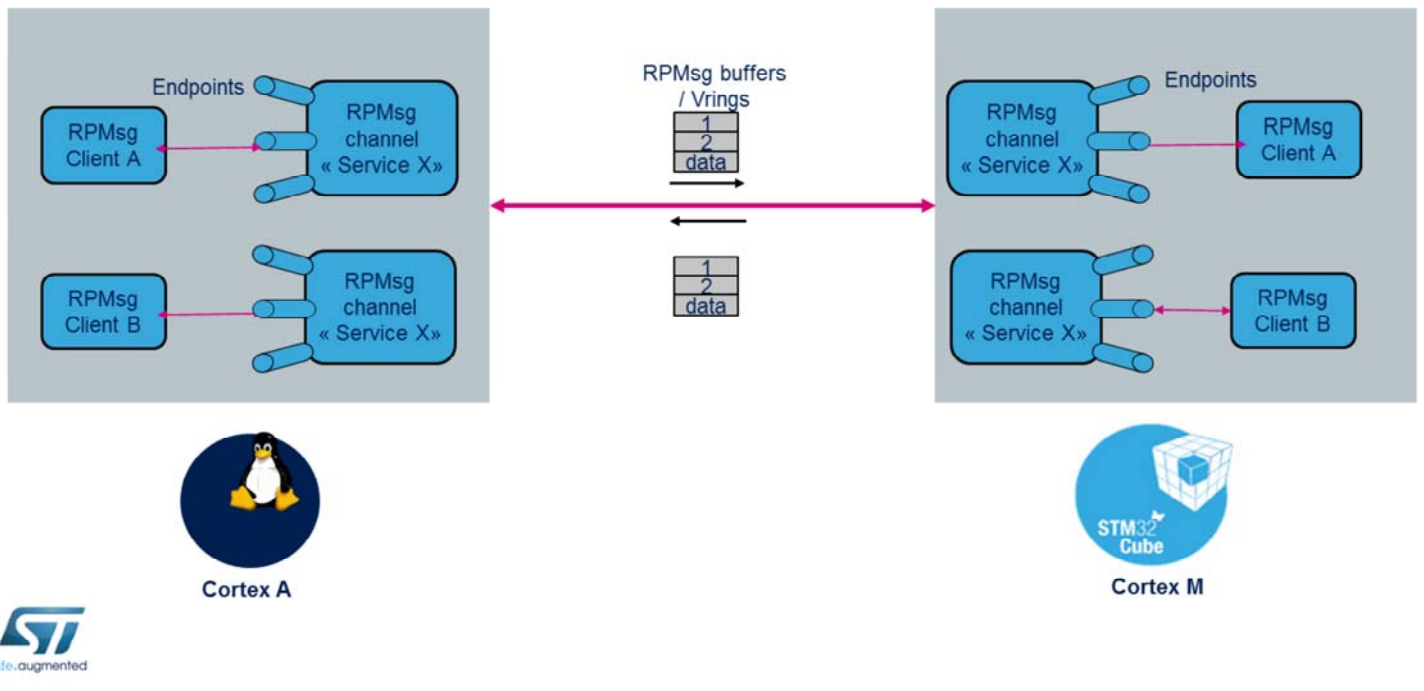
The RPMsg messaging uses 2 bidirectional channels. A channel is used in one direction to inform that a RPMsg buffer is available and in the other direction to inform than the RPMsg buffer has been released.

In the STM32 MPU Embedded Software distribution, the

IPCC channel 1 is used for messages from Cortex-M4 to Cortex-A7 core and the IPCC channel 2 is used for messages from Cortex-A7 to Cortex-M4 core.

On Cortex-A7, IPCC is controlled by the mailbox software framework.

On Cortex-M4, IPCC is controlled by the HAL_IPCC software driver.



In the STM32 MPU Embedded Software distribution, the RPMsg protocol is used for the inter-processor communication.

This slide is a presentation of the RPMsg protocol and its associated concepts.

The RPMMSG protocol consists in sending a message from a local address to a remote address. The message is stored in a buffer in the shared memory. The Virtio layer is in charge of the management of the buffer lifecycle using ring buffers with associated descriptors named Vrings.

On each Cortex, a RPMsg client offers a service. The client is identified by:

- Its service, defined by a service name
- Its endpoint, defined by an address identifier and operating callbacks.

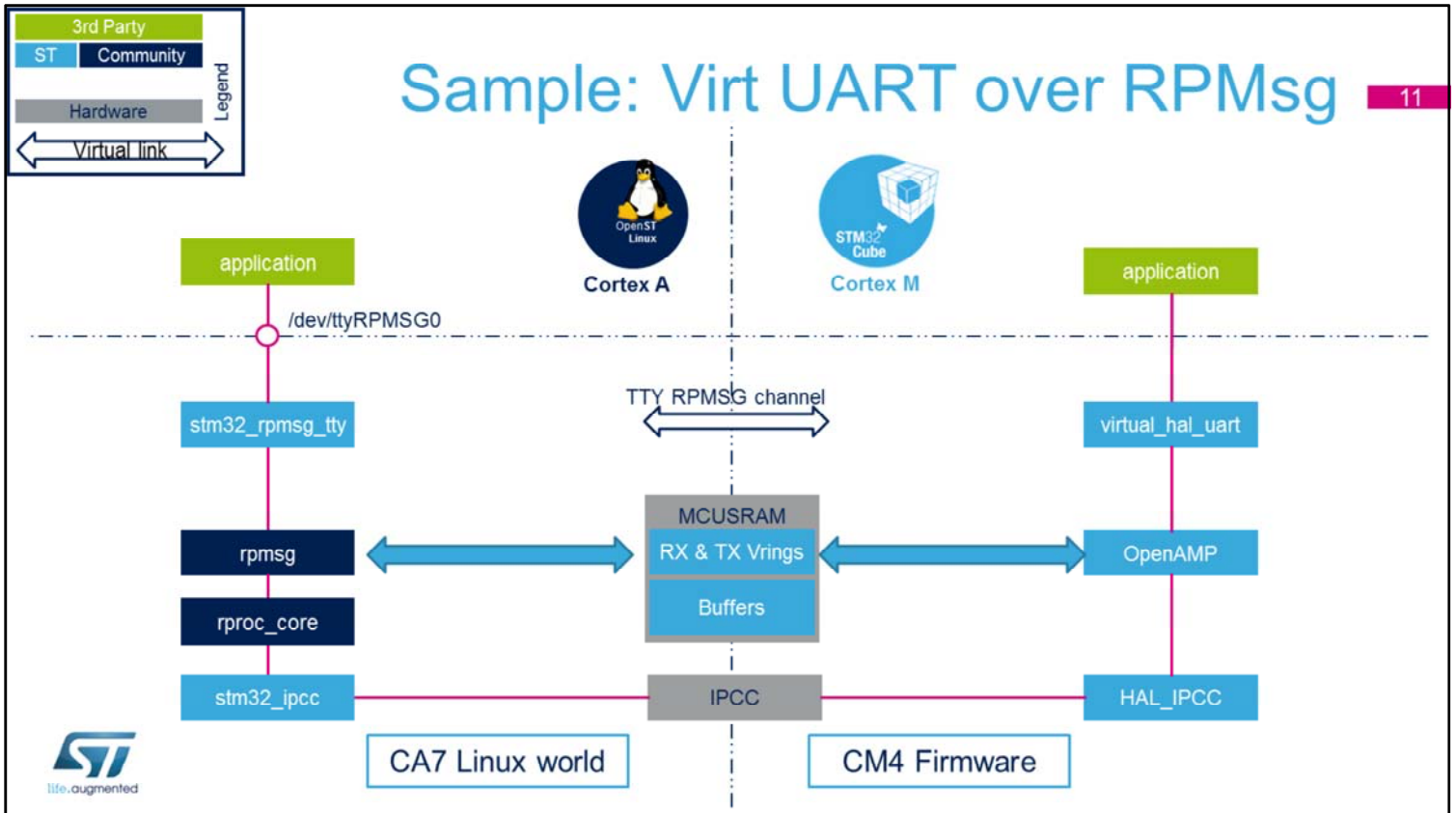
On service registration, the RPMsg framework sends a «name service announcement» message to the remote processor. In this message the address of the client endpoint

is provided.

On remote processor side, the RPMSG framework checks if a local client has register the same service. In this case a channel is created between both RPMsg clients and the local RPMsg client is informed that a channel is bound.

Sample: Virt UART over RPMsg

11



The virtual UART is an example implemented for STM32MP1 devices to demonstrate an application using the RPMsg protocol. The aim here is to simulate an UART over the RPMsg protocol.

On the Cortex-A7 core, a `stm32_rpmsg_tty` driver is a RPMsg client, implemented to expose to Linux user-land a serial TTY console, and to perform the adaptation layer between the UART and the RPMsg protocol.

On the Cortex-M4 core, a “virtual HAL UART” service is the RPMsg client. It provides a HAL UART API for application and also provides the same adaptation layer between the UART and the RPMsg protocol.

The RPMMSG channel is created by the initialization of the Virtual UART on Cortex-M4. This trigs the creation of an endpoint for the “TTY RPMMSG” service, and the sending of the “new service” announcement to the Cortex-A7.

On the Cortex-A7 side, the message is processed and the

corresponding service is associated to the `stm32_rmpsg_tty` driver. The Linux driver is probed to create an endpoint and a `/dev/ttyRPMMSG0` file system interface is created. From now on, the Cortex-A7 core is able to send messages through the virtual UART channel.

Notice that it is possible to create several instances of the virtual UART. In term of RPMMSG protocol, this action consists in creating a new endpoint per instance. The result is the creation on the Cortex-A7 side of a `/dev/ttyRPMMSG<X>` interface.

- To understand STM32 resource management mechanism, there are a few terms to be familiar with:
 - **Assignment**: a peripheral assignment to a core means that this peripheral (or Hardware block) is used by that processor.
 - **Shared resource**: shared or system resource required to operate a peripheral, and controlled by the master core the peripheral is assigned to (clocks, regulators, gpios...)



The coprocessor resources management corresponds to the implementation of a mechanism for the management of the peripherals in a multicore systems.

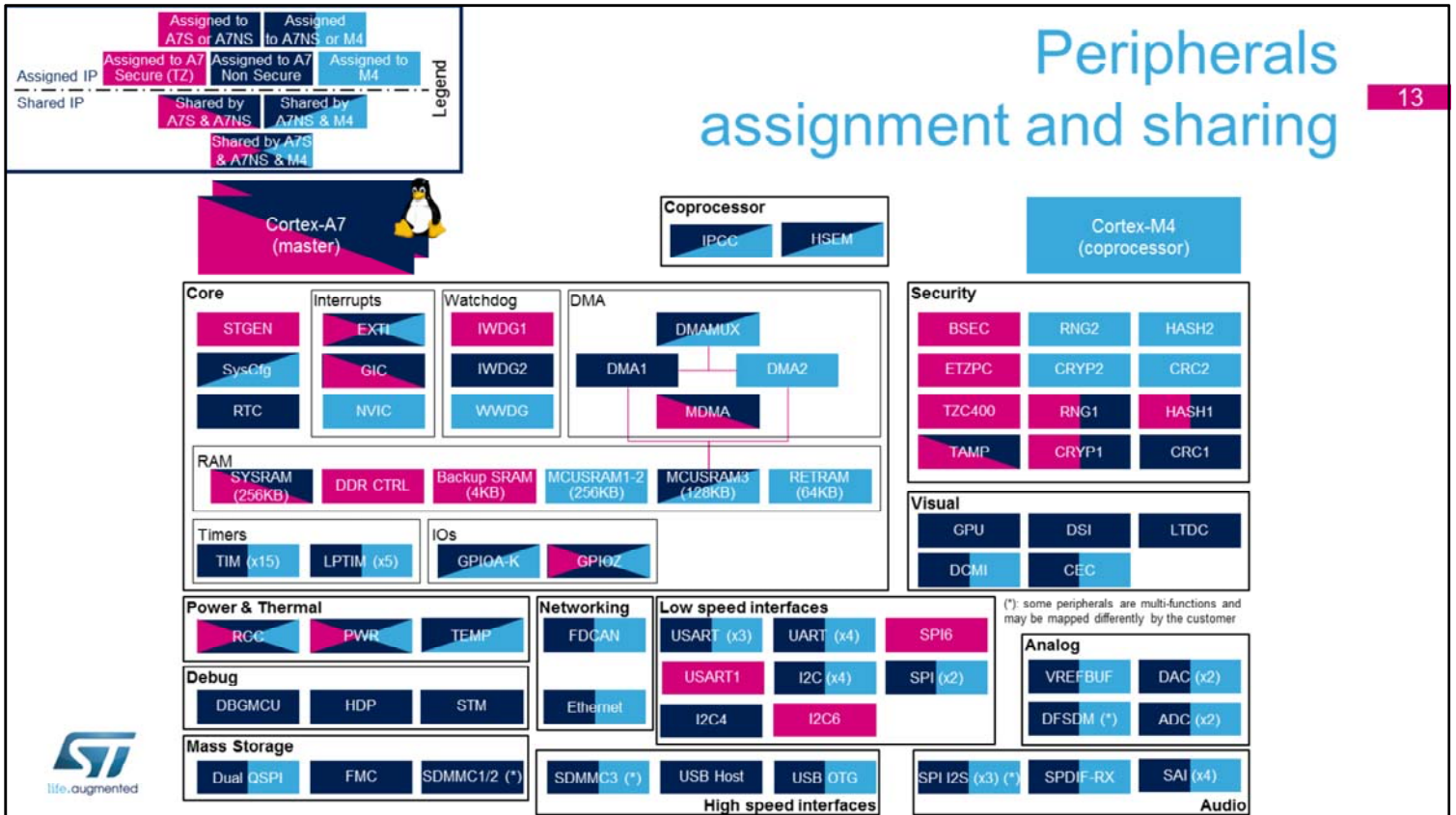
First of all, to understand the resource management, some concepts are to be defined.

The term peripheral assignment means the action to assign a set of peripherals to a cortex context. For instance an I2C peripheral can be assigned to either the Cortex-A7 or the Cortex-M4 core .

At the oposite, some peripherals or resources must be shared accross several contexts. The shared resources are typically system resources, like RCC for the reset and clock control.

Peripherals assignment and sharing

13



This slide presents the possible peripherals assignment on the STM32MP1 platform.

In line with this concept, 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 be done with the STM32CubeMX tool or manually, and it is reinforced by hardware isolation for the Cortex-A7 secure and for the Cortex-M4 core.
- “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 ensure no contention can happen 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.
- The RCC block, just below, is a system peripheral that can be concurrently accessed by the three runtime contexts.

The purpose of the followings slides is to explain the concepts implemented on the STM32MP1 platform for the management of the “assigned” and “shared” resources.

All the mechanisms involved in the assignments are further described in ST Wiki pages.

Note that this diagram shows the STMicroelectronics recommendations or choices of assignment in the STM32 MPU Embedded Software distribution. Additional possibilities might be described in the STM32MP1 reference manual and may be considered later on in this distribution.

Peripherals management challenges

14

- Assign a peripheral to Cortex-M4 or Cortex-A7 core:
 - Means that a peripheral (or Hardware block) is used only by that processor.
 - Developer is responsible for the coherency of the assignment in the global system.
 - The STM32CubeMX tool can do it for developer by populating device trees in TF-A, Uboot, Linux and HAL_init for Cortex-A7 or Cortex-M4 usage.
- Isolation:
 - A peripheral can be isolated for secure part or for Cortex-M4 core.
 - TF-A secure firmware manages the isolation of the peripheral based ETZPC registers according to its device tree.
 - The STM32CubeMX tool can do this for developer by populating the Uboot/TF-A Device tree.
- Configure shared resources like Clocks and GPIOs.
 - The resource is either protected by a hardware semaphore or configured by Linux according to the device tree.
 - The STM32CubeMX tool can do this for developer by populating the Linux Device tree and the HAL_init function



To understand the implementation of the resources management, it is important to understand the associated challenges arising in a multi-processor environment.

On STM32MP1 microprocessors, two contexts are running in parallel. When a peripheral is assigned to one context, this peripheral has to be managed only by this context. So the coherency of the global system has to be ensured. To achieve this, the STM32CubeMX tool offers an interface to assign the peripherals.

The STM32MP1 offers also the possibility to isolate a peripheral to the secure part or the Cortex-M4. The isolation is managed by the secure firmware TF-A through the Extended TrustZone Protection Controller (ETZPC) table configuration. The STM32CubeMX tool can also manage this by configuring the TF-A Device tree involved for the isolation.

Lastly, some of the resources are shared resources. These resources are generally required to operate the peripheral. Depending on its type, the system resources have either to be protected by an hardware semaphore or exclusively managed by the Linux Context to ensure coherency in the system (for instance the clock tree management). In the same way, the STM32CubeMX tool can help to correctly configure the shared resources.

Peripherals assignment via STM32CubeMX

15



• When Checking the box:

- Code is generated in TF-A Device tree to isolate the peripheral to one of the contexts:
 - Cortex-A7 secure
 - Cortex-M4
 - Cortex-A7 non-secure.
- Code is generated to enable the driver on Cortex-A7 or Cortex-M4 (Device tree).
- Code is generated in Linux Device tree to declare all shared resources needed to operate the peripheral (clock, regulator, GPIOs, EXTI) depending on peripheral configuration.
- Cube firmware code is generated to initialize the peripheral associated to the Cortex-M4 core.

This screenshot shows the example of an assignment as it can be done with the STM32CubeMX tool:

- USART4 is assigned to the Cortex-A7 non-secure context for Linux
- USART5 is assigned to the Cortex-A4 context for STM32Cube

Depending on the configuration selected, the STM32CubeMX tools assigns the peripheral to a cortex context by:

- Isolating the peripheral by configuring the TF-A Device tree,
- Configuring the Linux Device tree to enable or disable the node depending on the assignment,
- Declaring the system resource involved to operate the peripherals assigned to the Cortex-M4 core,
- Generating cube Firmware initialization code for the peripheral assigned to the Cortex-M4 core.

- To help ensure the coexistence of two execution contexts, the resource management provides several services:
 - Peripheral assignment request: Mechanism used on Cortex-M4 to ensure that a peripheral is reserved for a processor use. The principle is to check the peripheral availability before starting to use it.
 - Coprocessor shared resource configuration set: Services available in the main processor (Cortex-A7 running Linux) to configure the shared resources needed to operate the peripheral on the coprocessor.
 - Shared peripheral protection with hardware semaphore: This service available on both processors enables the protection of the GPIOs and EXTI shared resources from concurrent accesses.
 - Dynamic shared resource reconfiguration: Services based on a remote messaging that allows the coprocessor to ask the main processor to update the shared resources configuration.



To help control and check the resource assignment, some services have been implemented in the STM32 MPU Embedded Software distribution.

- The first service is the peripheral assignment check implemented in the STM32Cube firmware. This utility relies on the ETZPC table and allows the check of the peripheral isolation before its initialization.
- The second service is the shared configuration set. Implemented on Cortex-A7 core, it allows the configuration of the clocks and regulators associated to a peripheral. These shared resources need to be managed by the Linux OS for the power management optimization.
- Then each Cortex firmware is in charge of the configuration of the GPIO and EXTI resources. The protection of these shared resources are ensured by the usage of the Hardware semaphore relying on the HSEM peripheral.
- Lastly a dynamic resource reconfiguration service has

been put in place to allow a coprocessor to change on the fly the clocks and regulator settings.

Peripheral assignment request 17

- Configuration based on ETZPC table :
During the boot stage, the trusted firmware TF-A configures the peripheral assignment:
 - TF-A reads the configuration from the bootloader Device tree
 - TF-A writes into the ETZPC registers.

Configuration	A7 Linux access	M4 firmware access
Secured	NO	NO
Non secured , M4 isolated	NO	YES
Non secured, A7 or M4	YES	YES
Non secured, A7 isolated *	YES	NO

* SW isolation only , no hardware isolation (although Hw access is possible)

- Check :

The peripheral assignment request feature allows the drivers to check for the peripheral accessibility. Before starting a driver, a peripheral access should be granted:

- On Cortex-A7: U-boot grants access by updating the Linux Device tree peripheral nodes relying on the ETZPC table.
- On Cortex-M4: Before starting a driver, a peripheral access should be granted by the ResourceManager utility.



The peripheral assignment request service relies on the bus firewall controlled by the Extended TrustZone protection controller (ETZPC) table.

Four modes are available depending on the secure/non-secure context and the assignment to the Cortex-A7 or Cortex-M4 contexts.

Note that the isolation of the non-secure Cortex-A7, at the opposite of the other modes, is not a real hardware isolation but a software protection, as accesses from the Cortex-M4 core are still possible.

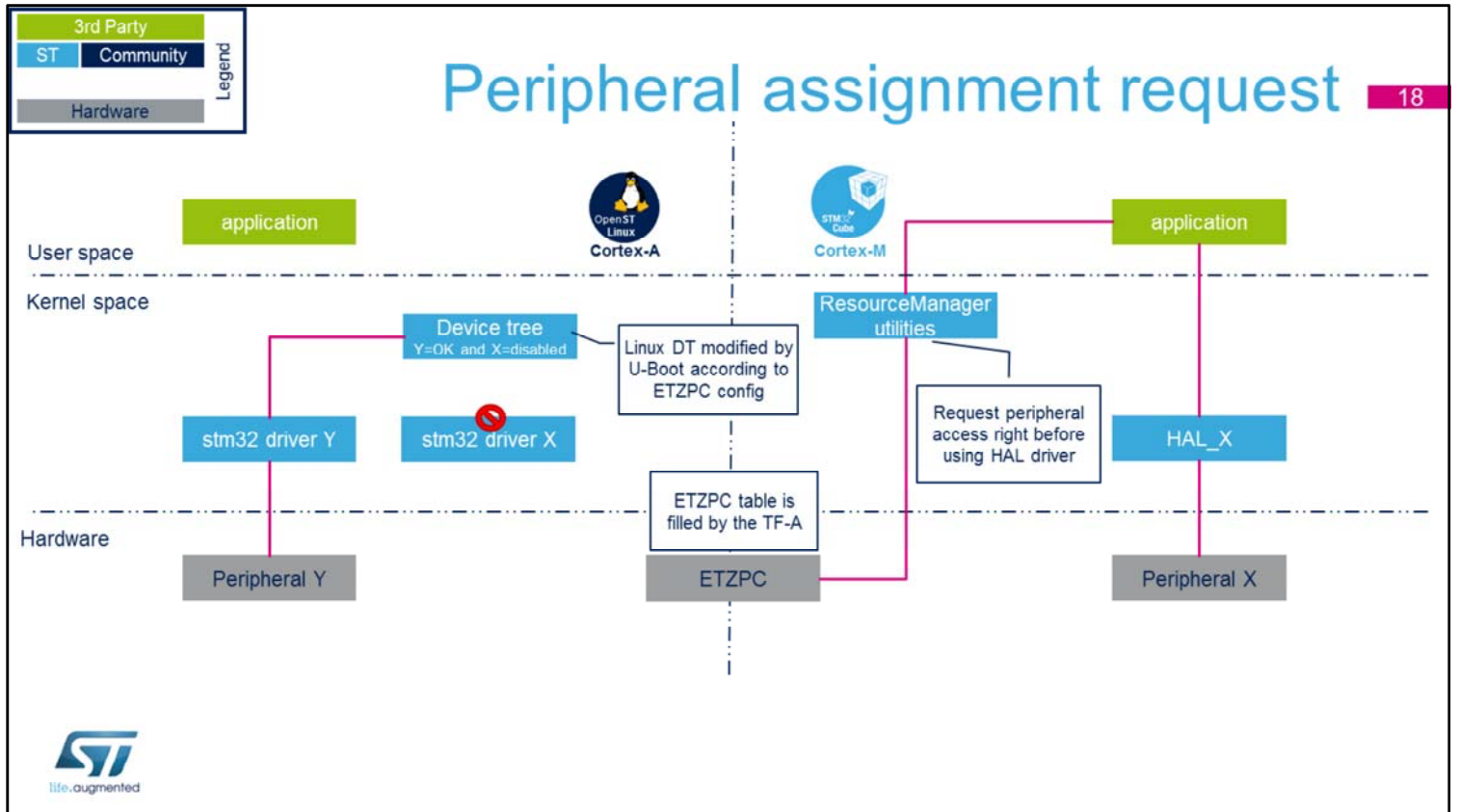
The ETZPC table is filled by the TF-A security firmware based on its Device tree.

Then during the initialization phase, an access is granted:

- On Cortex-A7 core, the U-Boot updates the Linux Device tree according to the ETZPC table. Depending on the

assignment, it enables or disables peripheral nodes declared to the Linux usage.

- On Cortex-M4 core, the application has to verify the peripheral assignment before initializing it. The resource manager utility provides services to grant this access.



This picture gives an overview of the software architecture related to the check of the peripheral assignment. In this example the TF-A firmware has isolated and assigned the peripheral-X to the Cortex-M4 core and the peripheral-Y to the Cortex-A7 core.

- On Cortex-A7 core, during the boot stage, U-Boot has enabled the peripheral-Y node and disabled the peripheral-X node. As a consequence, only the stm32 driver for the peripheral-Y is enabled.
- On Cortex-M4, the application grants the access right to the peripheral-X thanks to the Resource Manger utility and then call the HAL API to initialize it.

Shared resource configuration set

19

- Shared resources must be protected from concurrent accesses when required by both cores to operate the peripherals. Two strategies:
 - The protection using an hardware semaphore
 - The management of the shared resource by only one core
- **Protection by hardware semaphore:** Default way to protect shared resources to avoid concurrent accesses to the resources registers.
 - **Pin:** the pin configuration is managed by the Linux pinctrl framework. In GPIO mode, the coprocessor can change the output value and read the input value.
 - **External interrupt (EXTI):** the bind between the GPIO and the associated EXTI interrupt is managed by the Linux regulator framework. The coprocessor is in charge of enabling/disabling the NVIC interrupt mask.
- **Management by the Cortex-A7 Linux Framework:** For specific system resources involved in power management strategy:
 - **Clock tree:** the clock configuration is managed by the Linux clock framework, the coprocessor is in charge of the clock gating for the peripheral.
 - **Regulator:** the power supply is managed by the Linux regulator framework.



The main purpose of this service is to correctly manage the configuration of the shared resources in the multi-core system.

To protect these resources, two strategies have been implemented:

- The protection of the shared resource by an hardware semaphore.
- The management of the shared resource by only one Cortex.

The protection by hardware semaphore, relying on the HSEM peripheral, is used by default. This consists in getting an hardware semaphore to grant exclusive access on the critical registers of a shared resource. This is the case for instance for the GPIOs and the EXTI configurations.

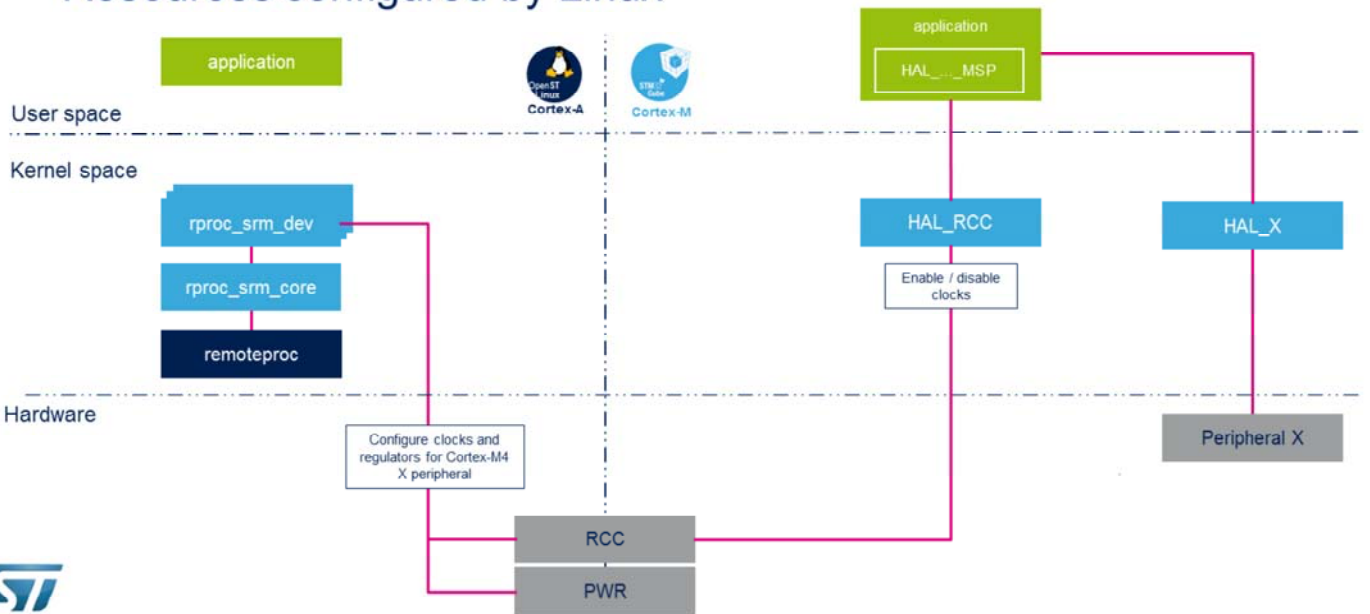
Some other shared resources like clocks and regulators, need to be managed by the Linux OS. The main reason is

the integrated power strategy natively implemented on Linux OS. Indeed, the clocks and regulators chaining can be represented by a tree, with parents and children nodes. This tree is monitored by the Linux OS to disable or enable parents depending on the state of their children. For instance Linux needs to be informed about clocks used by the Cortex-M4 core. Without this information, it considers that the clock is not in use and could stop the PLL clock as the parent node.

Shared resource configuration set

20

- Resources configured by Linux



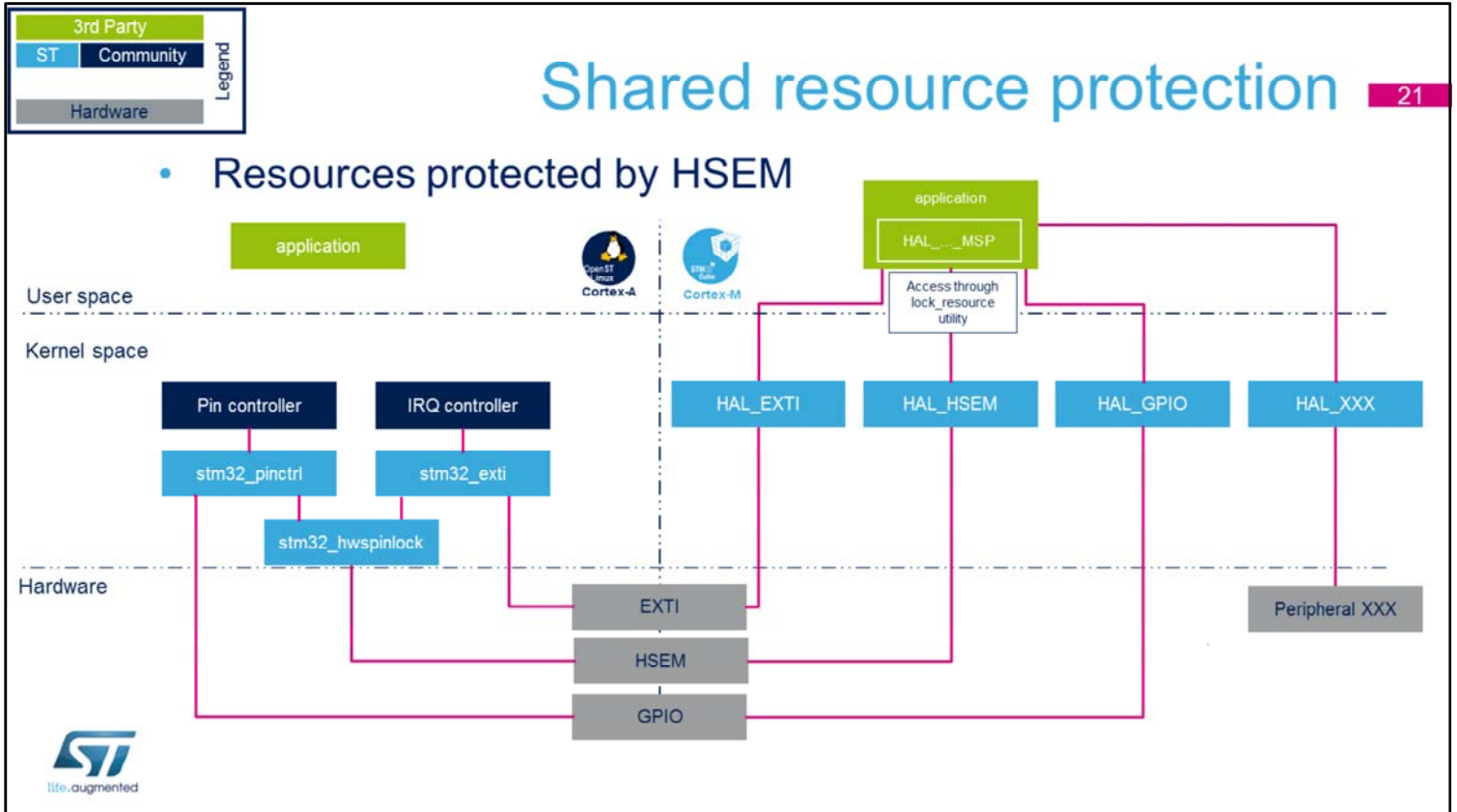
This slide shows the software framework involved in the configuration of the shared resources by the Linux OS on Cortex-A7 core.

The Remote PROCessor System Resource Manager (RPROC SRM) framework has been developed by ST to declare and configure shared resources used by the Cortex-M4 core to operate the assigned peripherals. This framework parses the Device tree for nodes associated to the Cortex-M4 peripheral and configure the shared resources accordingly. A SRM device driver instance is enabled per sub-node.

On Cortex-M4 side, the clock and regulator resources are not initialized. Only the peripheral clock is gated as clock gating registers are Cortex dependent.

Shared resource protection

21



This slide shows the software framework involved in the protection of shared resources by the hardware semaphore. On cortex-A7 core, accesses to the GPIOs and EXTI configuration registers are protected by the hardware spinlock that gets a HSEM semaphore if free or wait for its release.

Please Note that the U-Boot bootloader also uses the same hardware semaphore during the boot stage.

On Cortex-M4 core, accesses protection has to be managed by the application, granted by the lock resource utility API. This utility offers a simple interface to lock/unlock a semaphore depending on the resource to protect.

Sample: I2C assigned to Cortex-M4

22



```
@soc {
  i2c1: i2c@40012000 {
    compatible = "st,stm32f7-i2c";
    status = "disabled";
  };
};

@m4_hw_resources {
  compatible = "rproc-srm-core";
  status = "enable";
  M4_IP_I2C@0x400012000 {
    compatible = "rproc-srm-dev";
    status = "okay";
    clocks = <&rcc_clk I2C1_K>;
    pinctrl-0 = <i2c1_pins_a>;
  };
};
```

Filled by
STM32CUBEMX

```
void HAL_I2C_MspInit(I2C_HandleTypeDef *hi2c)
{
  /*HW semaphore Clock enable*/
  __HAL_RCC_HSEM_CLK_ENABLE();

  if (SystemResourceConfigAllowed(SYS_RES_RCC)) {
    /*##-1- Enable the HSI clock */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
      /* Error */
      while(1);
    }

    /*##-2- Configure HSI as I2C clock source */
    RCC_PeriphCLKInitStruct.PeriphClockSelection = RCC_PERIPHCLK_I2Cx;
    RCC_PeriphCLKInitStruct.I2c123ClockSelection = RCC_I2CxCLKSOURCE;
    HAL_RCCEx_PeriphCLKConfig(&RCC_PeriphCLKInitStruct);

    /*##-3- Enable peripherals and GPIO Clocks #####*/
    /* Enable GPIO TX/RX clock */
    I2Cx_SCL_GPIO_CLK_ENABLE();
    I2Cx_SDA_GPIO_CLK_ENABLE();

    /* Enable I2Cx clock */
    I2Cx_CLK_ENABLE();

    /*##-4- Configure peripheral GPIO #####*/

    PERIPH_LOCK(GPIO);
    HAL_GPIO_Init(I2Cx_SCL_GPIO_PORT, &GPIO_InitStruct);
    HAL_GPIO_Init(I2Cx_SDA_GPIO_PORT, &GPIO_InitStruct);
    PERIPH_UNLOCK(GPIO);
  }
}
```



Regarding the code configuration, the STM32CubeMX tool generates the code to manage the shared resources. This slide describes the generation of the shared resources for an I2C port assigned to the Cortex-M4 core.

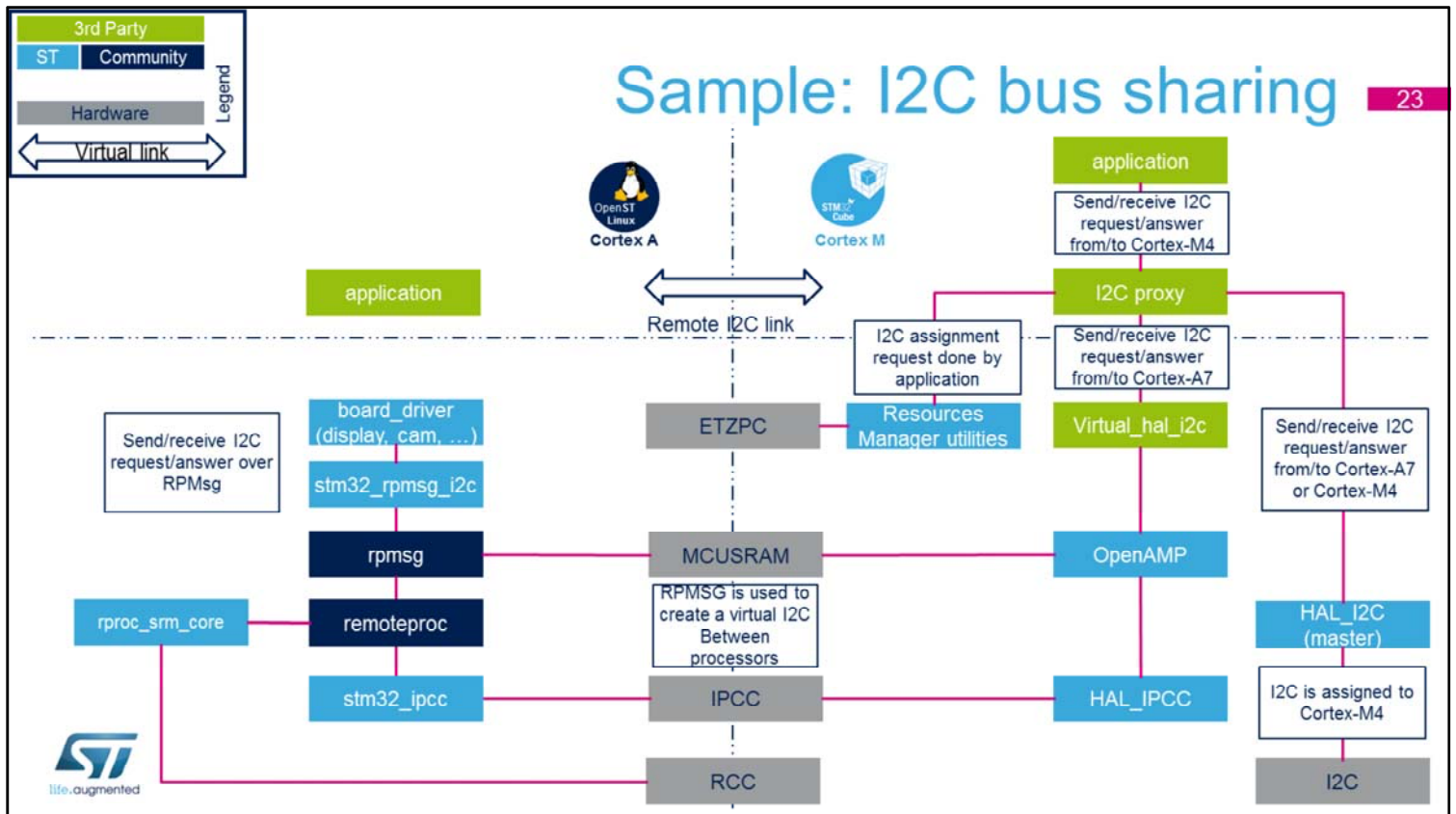
In the Cortex-A7 Device tree, the resource manager node is declared with an I2C1 sub node defined with its associated clock and GPIOs declaration. Please note that the I2C1 node used to assign the peripheral to the Cortex-A7 core has been previously disabled. Note also that the pins used for the I2C port are also declared in the Device tree. This part is optional as GPIOs are protected by an hardware semaphore. But this declaration can be used for debug to allow Linux OS to crosscheck that pins are not reserved for another peripheral.

On Cortex-M4 side:

- The clock initialization is protected by the

“SystemResourceConfigAllowed” service. Indeed in some specific cases, the Cortex-M4 core may have to initialize the clock tree. This is the case if the Cortex-M4 is loaded by the U-boot stage and so started before the Linux Firmware. Using this service allows the use the same firmware independently from the booting mode.

- In this example, the GPIO configurations are protected by a lock service that gets an hardware semaphore.



Here is a more complex example using most of the concepts previously presented.

It explains how to share an I2C port between the Cortex-A7 and the Cortex-M4 cores on an STM32MP1 platform.

In this example, the I2C peripheral is managed by the Cortex-M4 core. This means that the I2C has been isolated for the Cortex-M4 thanks to the ETZPC table and that the Linux Firmware SRM framework has configured the clocks needed to operate the I2C bus.

To transfer the I2C flows between the Cortex-A7 and the Cortex-M4 cores, a virtual I2C link over RPMsg can be implemented, with the same principle as the virtual UART proposed in the STM32 MPU Embedded Software distribution.

On the Cortex-M4 core, an I2C proxy is required to enable the sharing of the I2C resources from the Cortex-M4

application or from the Virtual I2C HAL.

For more details on coprocessor management or on the ST32MP1 product itself, please refer to the STM32MP1 user guide available on st.com.