i.MX 7

A detailed look on the heterogeneous architecture

FTF 2016 - 04/19/2016

Jean-Baptiste Théou

Sr. Embedded Software Engineer

TABLE OF CONTENTS

Introduction

ABOUT THE AUTHOR

- Jean-Baptiste Théou
- Senior Embedded Software Engineer @ **Adeneo Embedded**
- Based in Seattle, WA

I.MX 7

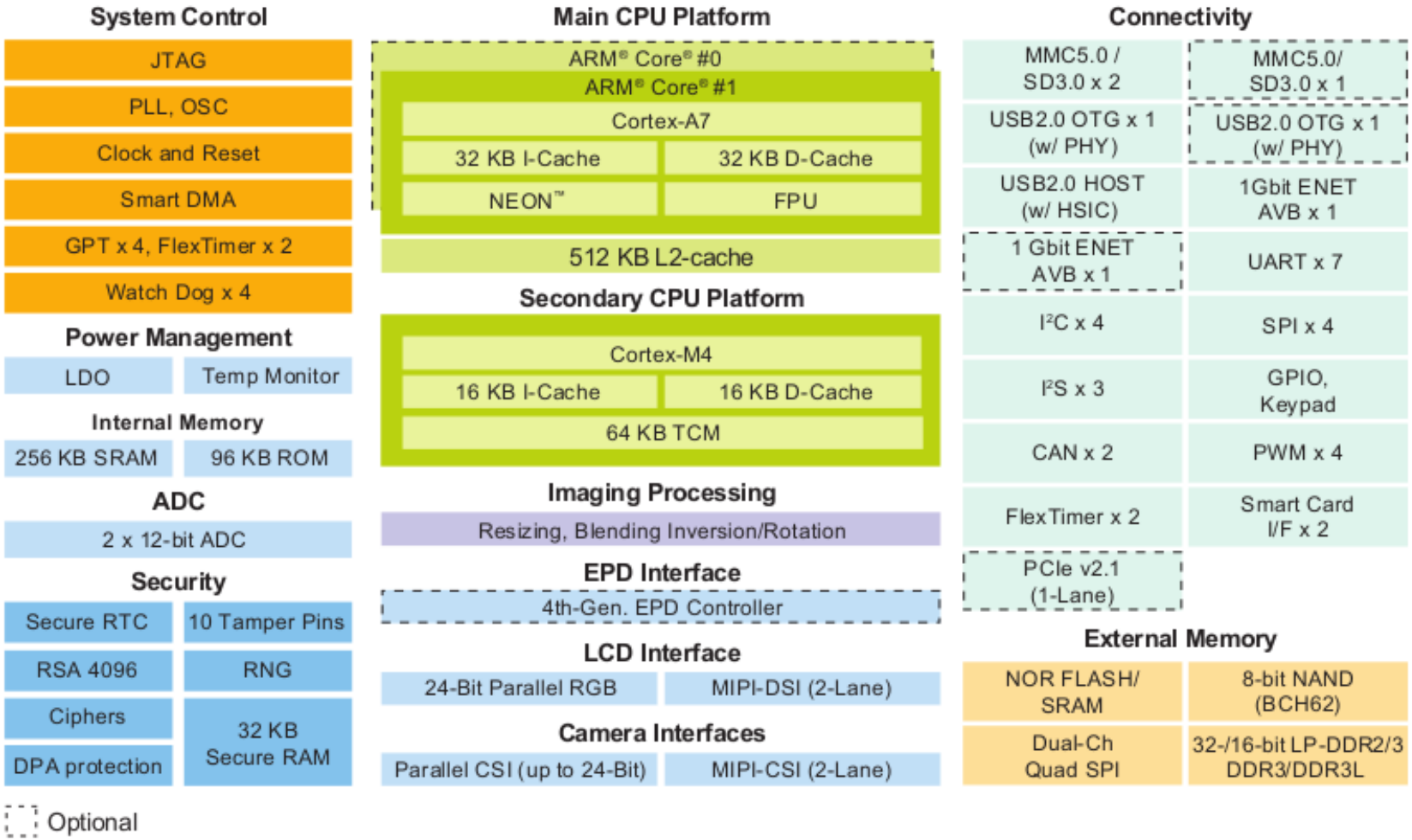
- Heterogeneous architecture
  - Cortex A7 (one or two cores)
  - One Cortex M4
- Advanced HW IP:
  - USB, Ethernet 1Gbit, CAN, LCD
  - 2x 12 bits ADC
  - PCIe (optional)
  - EPD controller (optional)
  - Security
  - And more!

I.MX 7

Architecture designed for:

- Power-efficiency
- Real-time application
- Complex application

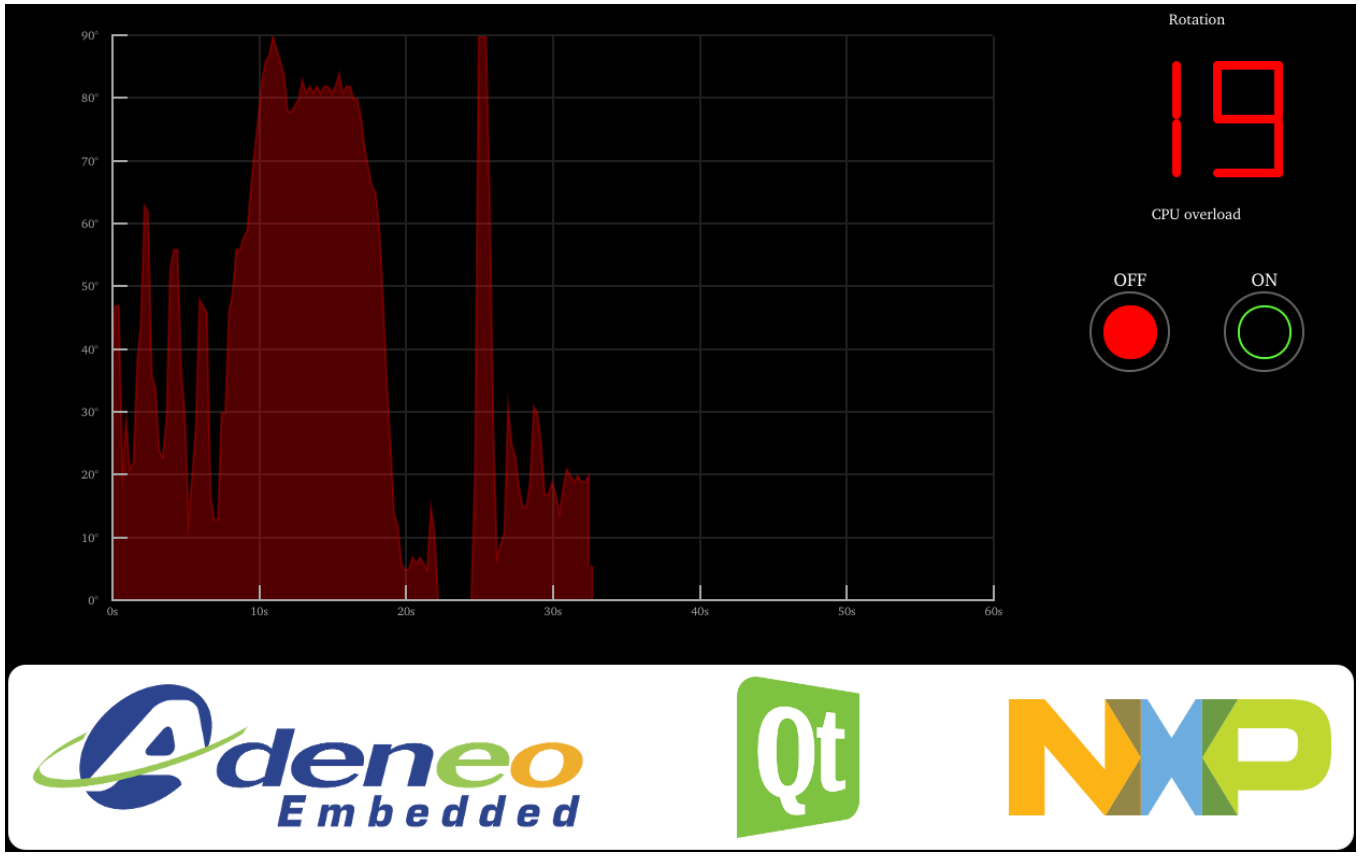
I.MX 7



Source: i.MX7 series - fact sheet

NO GPU, WHAT ABOUT UI?

The i.MX 7 doesn't have a GPU, but this doesn't mean you cannot have an evolved UI.



NO GPU, WHAT ABOUT UI?

One of the ways is to rely on Qt Quick 2D Renderer.

- Allow QT OpenGL app to run on GPU-less devices
- No need to rewrite code for QtQuick applications.
- Can use any 2D acceleration available on the device
- Several limitations. See QT website for more info

Why i.MX 7?

YOUR PRODUCT

Your product requirements are:

- I want power consumption efficiency
- I want fancy features: Communication (WiFi, BT, etc), UI, name it.
- I want real-time capabilities.
- I want runtime robustness.

WITH I.MX7

The i.MX 7 is the perfect match for your needs:

- I want power consumption efficiency.

⇒ **HW designed for this purpose**

- I want "fancy" features: Communication (WiFi, BT, etc), UI, name it.

⇒ **Linux on Cortex A7**

- I want real time capabilities.

⇒ **RTOS on Cortex M4**

- I want runtime robustness.

⇒ **HW designed for this purpose**

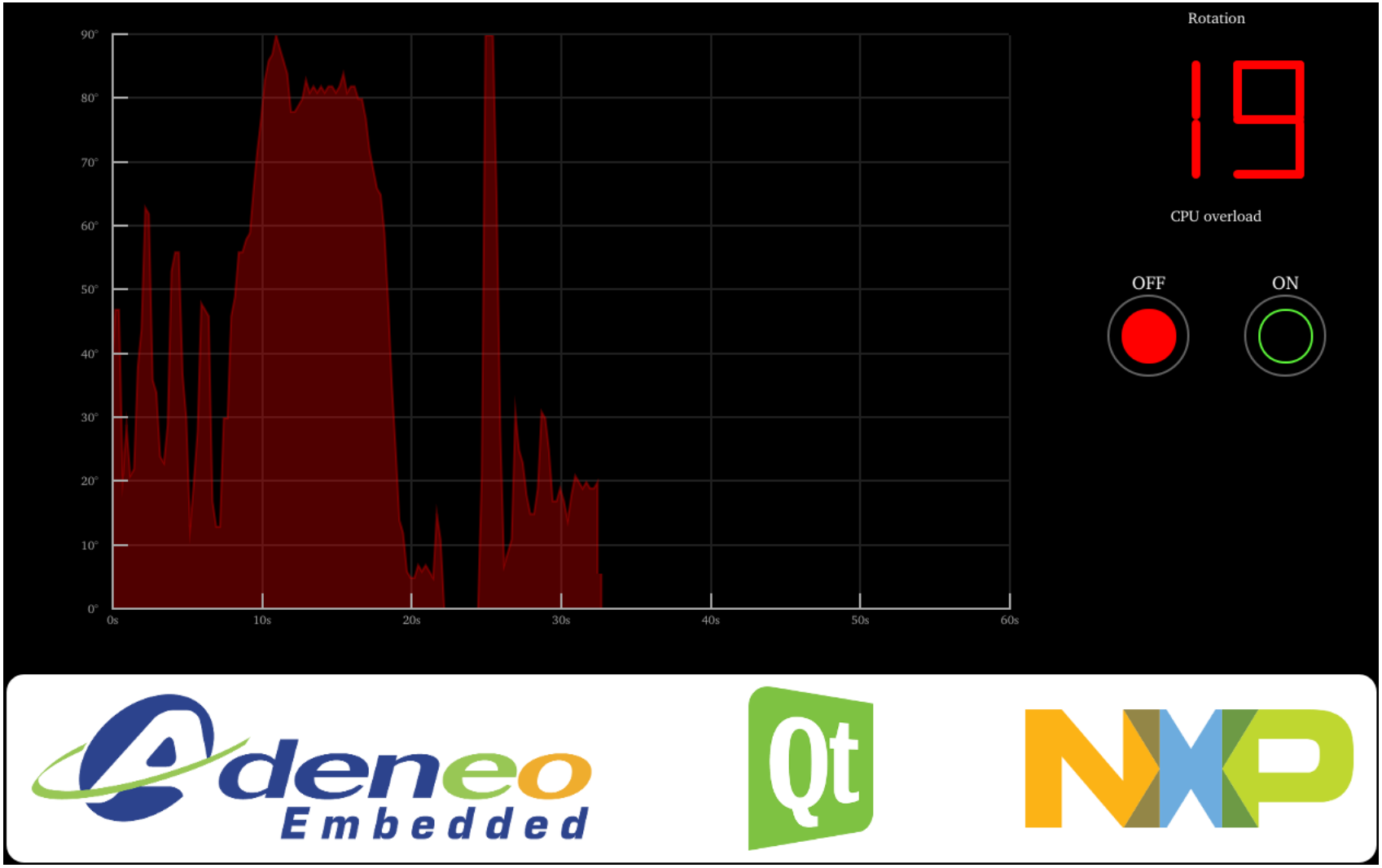
Use-case: Our demo

OUR DEMO

Our demo displays the advantage of the i.MX7 on the reference platform.

- "Fancy" features: Qt Quick2 application with cloud capability
- Real-time processing placeholder
  - Reading accelerometer value with the M4
- Load CPU "feature": Highlights the need of separation between the UI and the critical task

OUR DEMO


90°
80°
70°
60°
50°
40°
30°
20°
10°
0°
0s
10s
20s
30s
40s
50s
60s
Rotation
19
CPU overload
OFF
ON
Adeneo Embedded
Qt
NXP

IMPLEMENTATION

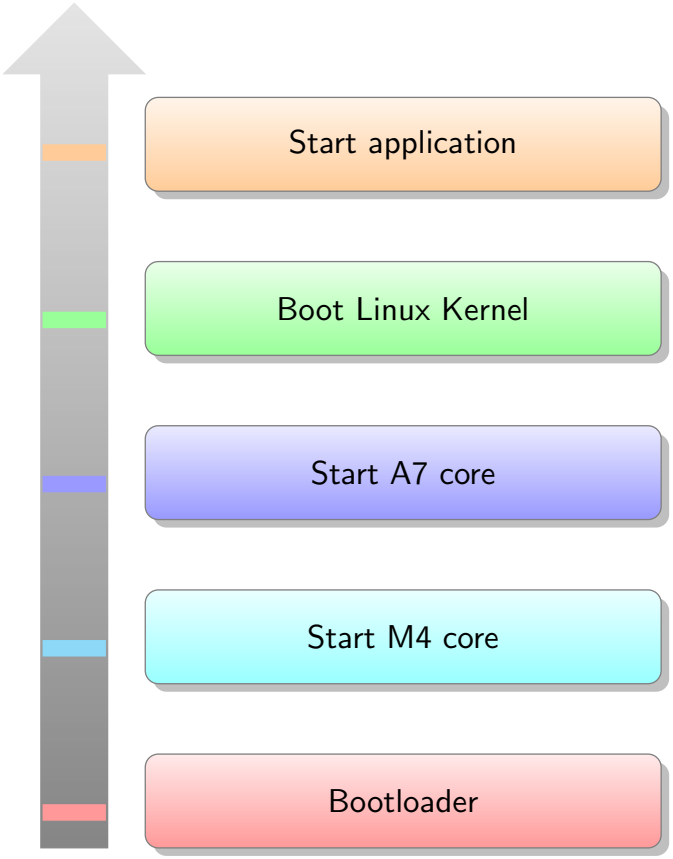
Our demo uses:

- FreeRTOS on Cortex M4
  - Read value from the accelerometer
  - Display the current value on a 7 segments (MikroBus)
  - Implement RPMSG communication to share the value with the A7
- Linux on Cortex A7 (Yocto)
  - Standard Linux
  - QT application, reading current accelerometer value with RPMSG

Take advantage of the i.MX 7

BOOT PROCESS IN I.MX7

Standard boot process of Linux, with a twist:

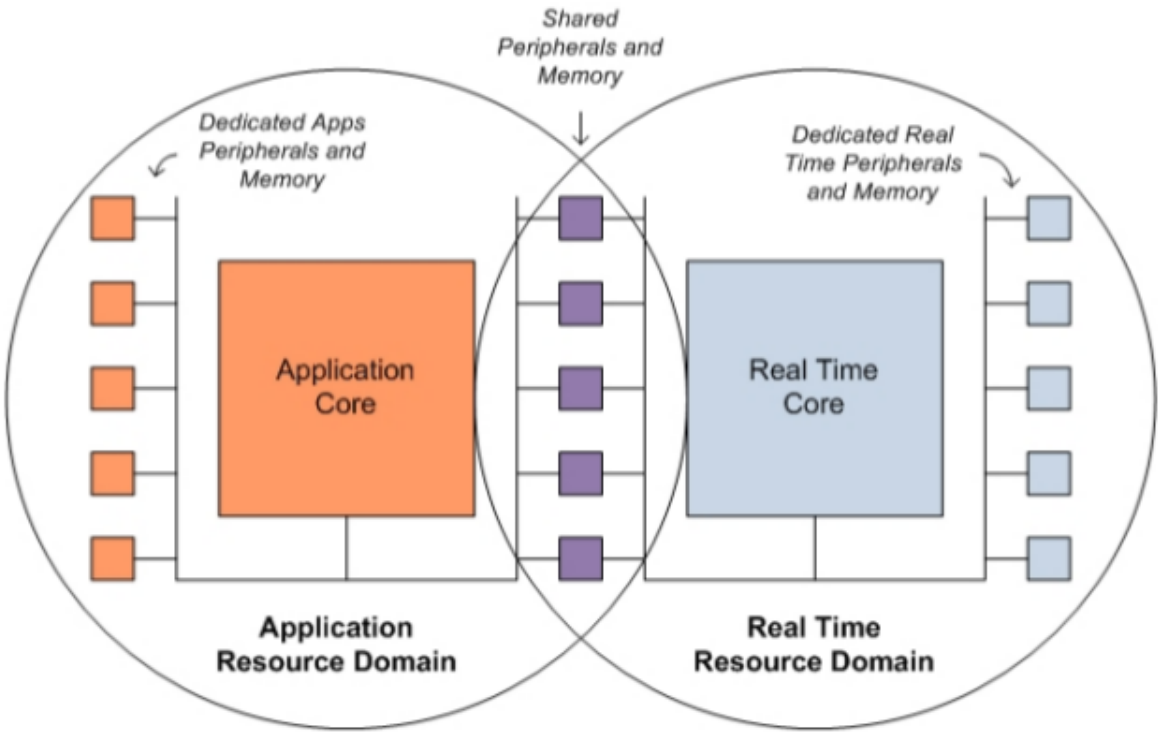


M4 AND A7 INTERACTION

- Master (A7)/Slave (M4) organization.
  - A7 starts clocks, write the firmware address information in the M4 bootROM and puts M4 out of reset
- M4 and A7 access the same peripherals
  - Resource Domain Controller: Ensures safe access to shared resources and allow access restrictions for peripherals/memory.
- RPMSG: Allow communication between the cores, using OpenAMP framework

RESOURCE DOMAIN CONTROLLER

Resource Domain Controller: Big advantage of the i.MX7.



Source: i.MX7 TRM

DEDICATED ACCESS

Dedicated peripherals and memory:

- Enhance security
- Remove programming error concerns
- Easy to implement

DEDICATED ACCESS

Different parts of the RDC (fine control):

- Domain ID (0-3): Set of cores, bus, IP
  - M4 Core, A7 Core, CSI, SDMA, uSDHC, etc
  - At reset, all of them are part of Domain ID 0.
- Peripheral Domain Access Permissions: Permission (R/W) for each domain
  - At reset, R/W allowed with no safe access for shared peripherals.
- Memory Region Control: Permission (R/W) for each domain, on a specific memory section.
  - Disabled at reset.

DEDICATED ACCESS: EXAMPLE

In our demo, we reserve I2C2 (accelerometer) and eCSPI3 (7Seg display) for the M4. The A7 cannot have access to it.

- Define Domain ID for M4 core to 1 (RDC_MDA1).
- Allow R/W permission for I2C2/eCSPI3 only for Domain ID 1 (RDC_PDAP67/RDC_PDAP100).
- And that's it! The I2C2/eCSPI3 accesses are now enforced on an HW level.

What about other chips on the buses?

In this situation, the use of hardware semaphore will be needed.

DEDICATED ACCESS: EXAMPLE

Use it!

Easy to implement on FreeRTOS, and it will make your design robust!

- Set the M4 on a specific domain with `RDC_SetDomainID`
- Inside your application, set the proper right to access the peripheral with `RDC_SetPdapAccess`
- You're done! For a more complex implementation, the code within FreeRTOS is a great start.

DEDICATED ACCESS: EXAMPLE

By default, the only peripheral locked for the M4 is the UART (board.c).

```
RDC_SetDomainID(RDC, rdcMdaM4, BOARD_DOMAIN_ID, false);
RDC_SetPdapAccess(RDC, BOARD_DEBUG_UART_RDC_PDAP, 3 << (
    BOARD_DOMAIN_ID * 2), false, false);
```

This command allows the UART to be used only by the M4, but it doesn't enforce the use of semaphore and doesn't lock the setting (can be changed from the A7).
To secure the I2C2 (and lock it) for the M4, just add:

```
RDC_SetPdapAccess(RDC, BOARD_I2C2_RDC_PDAP, 3 << (BOARD_DOMAIN_ID
     * 2), false, true);
```

DEDICATED ACCESS: EXAMPLE

In our application, we are enforcing HW access:

- Domain 1 R/W for I2C2 and eCSPI3

```
RDC_SetPdapAccess(RDC, BOARD_I2C_RDC_PDAP, (3 << (BOARD_DOMAIN_ID
      * 2)), false, true);
RDC_SetPdapAccess(RDC, BOARD_ECSPI_MASTER_RDC_PDAP, 3 << (
     BOARD_DOMAIN_ID * 2), false, true);
```

Source: commons.wikimedia.org

SHARED ACCESS

For the shared peripherals:

- Safe sharing: HW semaphore to ensure exclusive access to peripherals.
- If the semaphore is enabled on a peripheral, the domain needs to lock the HW semaphore before accessing the peripheral.
- Disabled by default.

I.MX 7 IMPLEMENTATION

- Two blocks of 64 gates each (RDC_SEMAPHORE(1/2)_GATE(1-64))
- One gate per peripheral
- Enable/Disable per peripheral, apply to all domains

I.MX7 IMPLEMENTATION

FreeRTOS provides an easy interface to control the access to a shared peripheral. For example, if we would like to share the access to the I2C2

- Enforce semaphore control (3rd argument to true).
  - `RDC_SetPdapAccess(RDC, BOARD_I2C_RDC_PDAP, (3 << (BOARD_DOMAIN_ID * 2)) | 0x2, true, true);`
- Use the API provided by FreeRTOS to lock/unlock the gate (The API find the block/gate)
  - `RDC_SEMAPHORE_Lock(BOARD_I2C_RDC_PDAP)`
  - `RDC_SEMAPHORE_Unlock(BOARD_I2C_RDC_PDAP)`
  - **Only the domain which locked the device can unlock it**
- On the Linux side, it would require to modify the driver to integrate the access control.

MULTICORE COMMUNICATION

RPMSG is used to send data between the cores. It's a part of the OpenAMP framework:

- Useful to export data out of the M4, like accelerometer values in our example
- Examples are available on Linux and FreeRTOS
  - Ping-Pong
  - String Echo
- They are a great start for custom implementation.

MULTICORE COMMUNICATION

In our demo, we are using a variation of the string echo example:

- Create an RPMSG channel for communication between the A7 and M4
- The A7 is the master, the M4 is the remote
- The A7 writes a command in the channel ('GET'), and the M4 answers back the value of the accelerometer

IMPLEMENTATION - FREERTOS

FreeRTOS contains the proper API to create/delete/read/write an RPMSG channel. You just need to implement your scenario:

- Create the channel of communication
- Try to get a new message and will block if none are available
- When we receive a message from the A7, the function unblock and we can process the message
- We copy the current value of the accelerometer into the output buffer allocated with:
- Send the answer to the A7
- Release the RX buffer
- Loop to the second step

IMPLEMENTATION - FREERTOS

Sample code (Check the source of FreeRTOS for more example)

```
rpmsg_rtos_init(0, &rdev, RPMSG_MASTER, &app_chnl);

for (;;) {
    rpmsg_rtos_recv_nocopy(app_chnl->rp_ept, &rx_buf, &len, &src,
     0xFFFFFFFF);
    /* Do some processing with the rx_buf */
    [...]
    /* Prepare the TX buffer */
    tx_buf = rpmsg_rtos_alloc_tx_buffer(app_chnl->rp_ept, &size);
    /* Send the value */
    rpmsg_rtos_send_nocopy(app_chnl->rp_ept, tx_buf, len, src);
    /* Release the RX buffer */
    rpmsg_rtos_recv_nocopy_free(app_chnl->rp_ept, rx_buf);
}
```

IMPLEMENTATION - LINUX

The proper API to create/delete/read/write an RPMSG channel is available inside the kernel. NXP provides an example to use a "fake" TTY device to communicate with the M4: `/dev/ttyRPMSG`

- Write 'GET' into the device (`/dev/ttyRPMSG`)
- Read the device to get the current value of the accelerometer

This implementation makes it very user friendly to communicate with the M4

MU: MESSAGE UNIT

- Share messages (events) between cores.
- Core-to-Core interrupts.
- Control power states between cores.

MU: MESSAGE UNIT

The current use of it is:

- M4 requests A7 to change clock parents.
- Share power state status between cores.
  - M4 shares its status with A7 : RUN, WAIT, STOP
  - Based on it, the A7 can go into Deep sleep mode (DSM)
- Implemented within FreeRTOS and Linux.
  - Check power consumption demo in the NXP booth: Using RPMSG and MU to demonstrate the different power states.

LINUX INTEGRATION

Linux kernel is "M4 aware":

- Peripherals dedicated to M4 are disabled by device (For M4-only peripherals, without modification, trying to access them will crash the module/kernel)
- MU driver in Linux `fsl,imx7d-mu`
- RPMSG driver in Linux `fsl,imx7d-rpmsg`

YOUR SCENARIO

- Out-of-box, everything already works.
- A lot of examples (including source) are available to start your own implementation (on FreeRTOS and Linux).
- Take the most out of this hybrid architecture!

Questions?