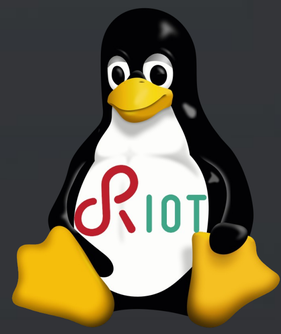
Towards fast-booting & MCU-driven operations for hybrid multi-core chips with dual Cortex-A7 and Cortex-M4

RIOT Summit
September 14 - 15, 2020

Gilles DOFFE
gilles.doffe@savoirfairelinux.com



STM32MP1 use case

Overview of STM32MP1

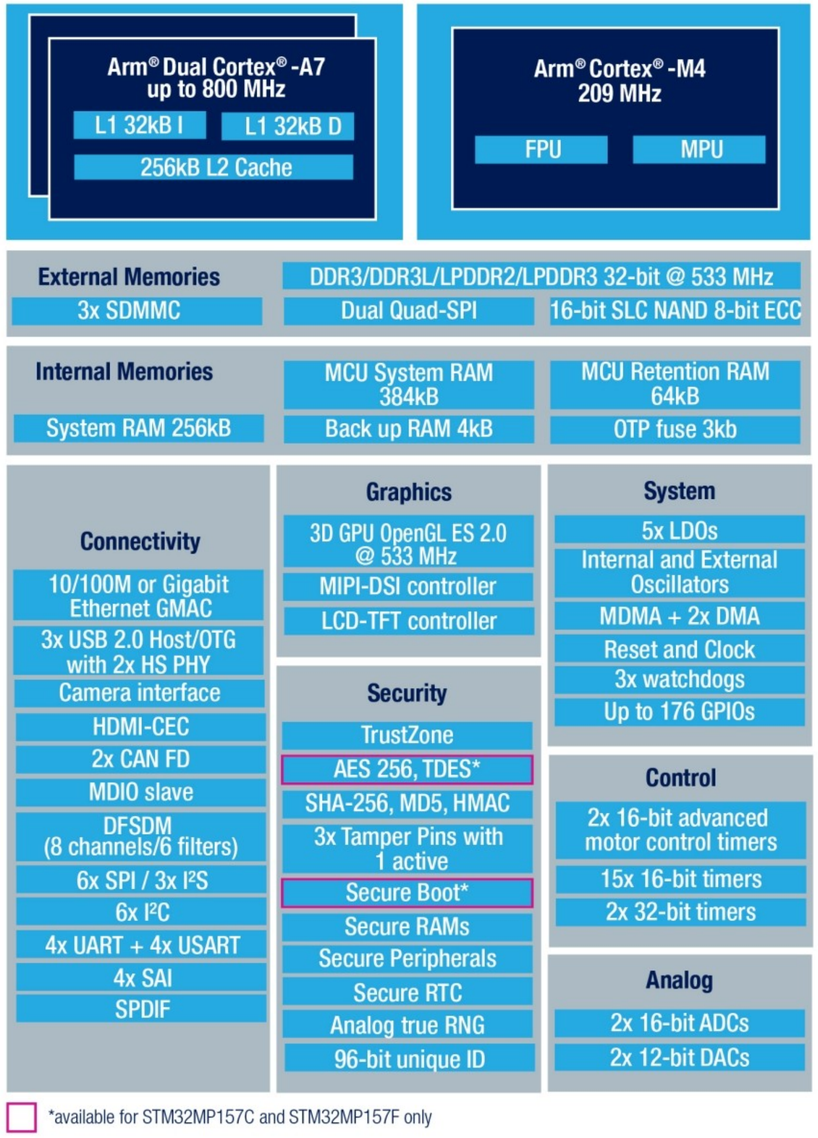
STM32MP157c-dk2 board description

Specificities

Boot modes

STM32MP1 family

- Let’s call it MP1 for convenience
- Use case of the STM32MP157CAC :
  - MP : Dual Cortex®-A7 cores running at 650 MHz
  - MC : Cortex®-M4 core running at 209 MHz
    - MPU/FPU
  - GPU
  - Several resources available :
    - Timers
    - GPIOs
    - I2C/SPI/UART
    - ADC/DAC
    - DMAs
    - RTC
  - Source and informations :
    - https://www.st.com/en/microcontrollers-microprocessors/stm32mp157c.html

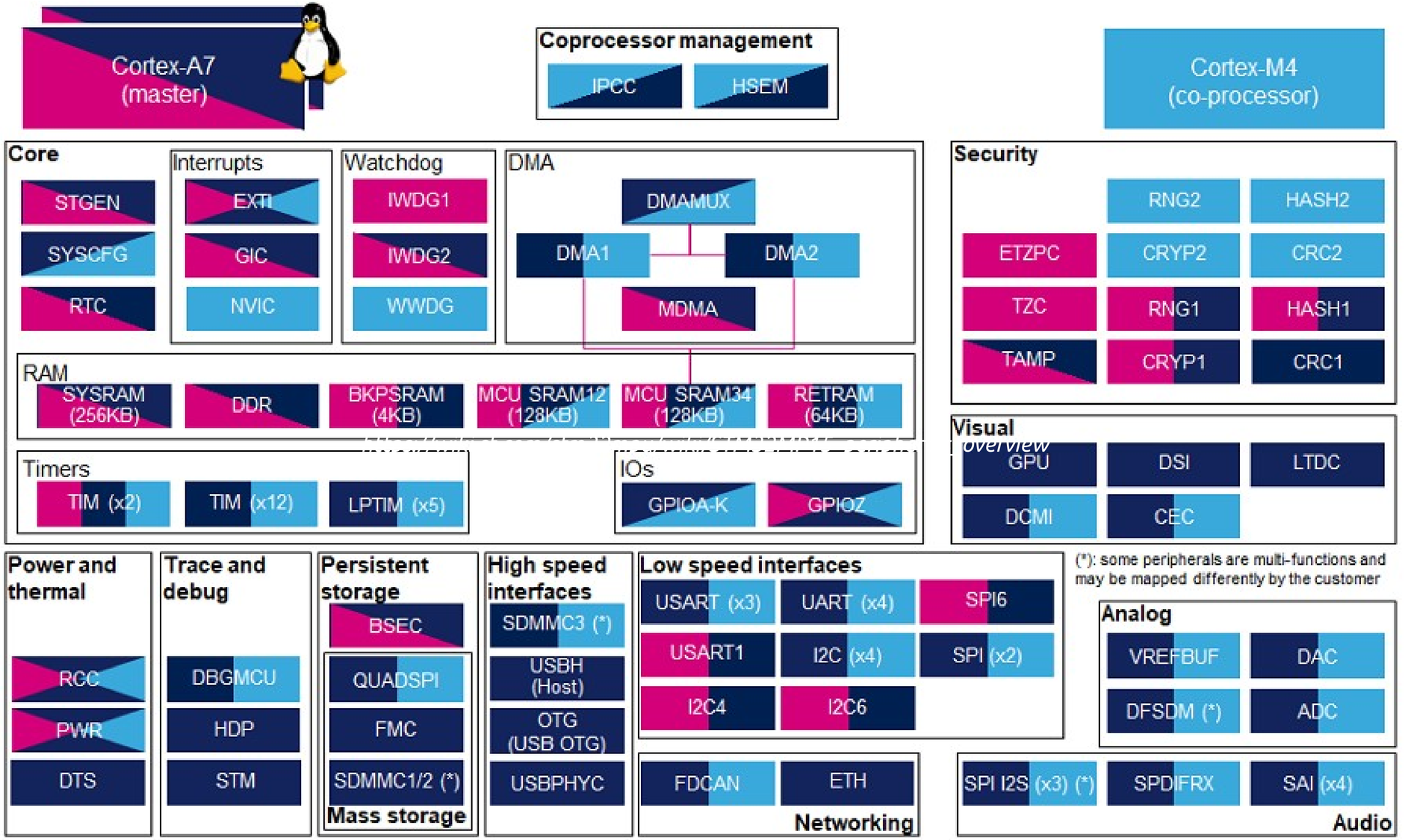


STM32MP157c-dk2 board

- Features
  - STM32MP157CAC with :
    - 4 Gb DDR3 clocked à 533MHz
    - Ethernet Gigabit interface
    - USB
    - 4 user LEDs
    - HDMI connector and Jack
    - Wi-Fi/Bluetooth
    - 4’’ TFT 480x800
  - Source and informations :
    - https://www.st.com/en/evaluation-tools/stm32mp157c-dk2.html

STM32MP1

Peripherals



› Source: https://wiki.st.com/stm32mpu/wiki/STM32MP15_peripherals_overview

Specificities

- Shared ressources
  - Can be protected using Hardware Semaphores (HSEM)
- No Flash, only SRAM
- MC (Cortex-M4) can be configured
  - by MP (Cortex-A7)
    - By bootloader (U-boot)
    - By Mainline Linux kernel (RCC, regulators and clocks mainly) and by itself
    - By OpenST Linux kernel (fully using Ressource Manager)
    - https://wiki.st.com/stm32mpu/index.php/Resource_manager_for_coprocessing#Principles
  - Fully by itself in a particular case

Boot modes

› 8 boot modes are available on STM32MP1 family :

BOOT2	BOOT1	BOOT0	Initial boot mode	Comments
0	0	0	UART and USB(1)	Wait incoming connection on: – USART2/3/6 and UART4/5/7/8 on default pins – USB High-Speed device on OTG_HS_DP/DM pins(2)
0	0	1	Serial NOR-Flash(3)	Serial NOR-Flash on QUADSPI(5)
0	1	0	eMMC™(3)	eMMC™ on SDMMC2 (default)(5)(6)
0	1	1	NAND-Flash(3)	SLC NAND-Flash on FMC
1	0	0	Reserved	Used to get debug access without boot from Flash(4)
1	0	1	SD-Card(3)	SD-Card on SDMMC1 (default)(5)(6)
1	1	0	UART and USB(1)(3)	Wait incoming connection on: – USART2/3/6 and UART4/5/7/8 on default pins – USB High-speed device on OTG_HS_DP/DM pins(2)
1	1	1	Serial NAND-Flash(3)	Serial NAND-Flash on QUADSPI (5)

Source :
Getting started with STM32MP151, STM32MP153 and STM32MP157 line hardware development

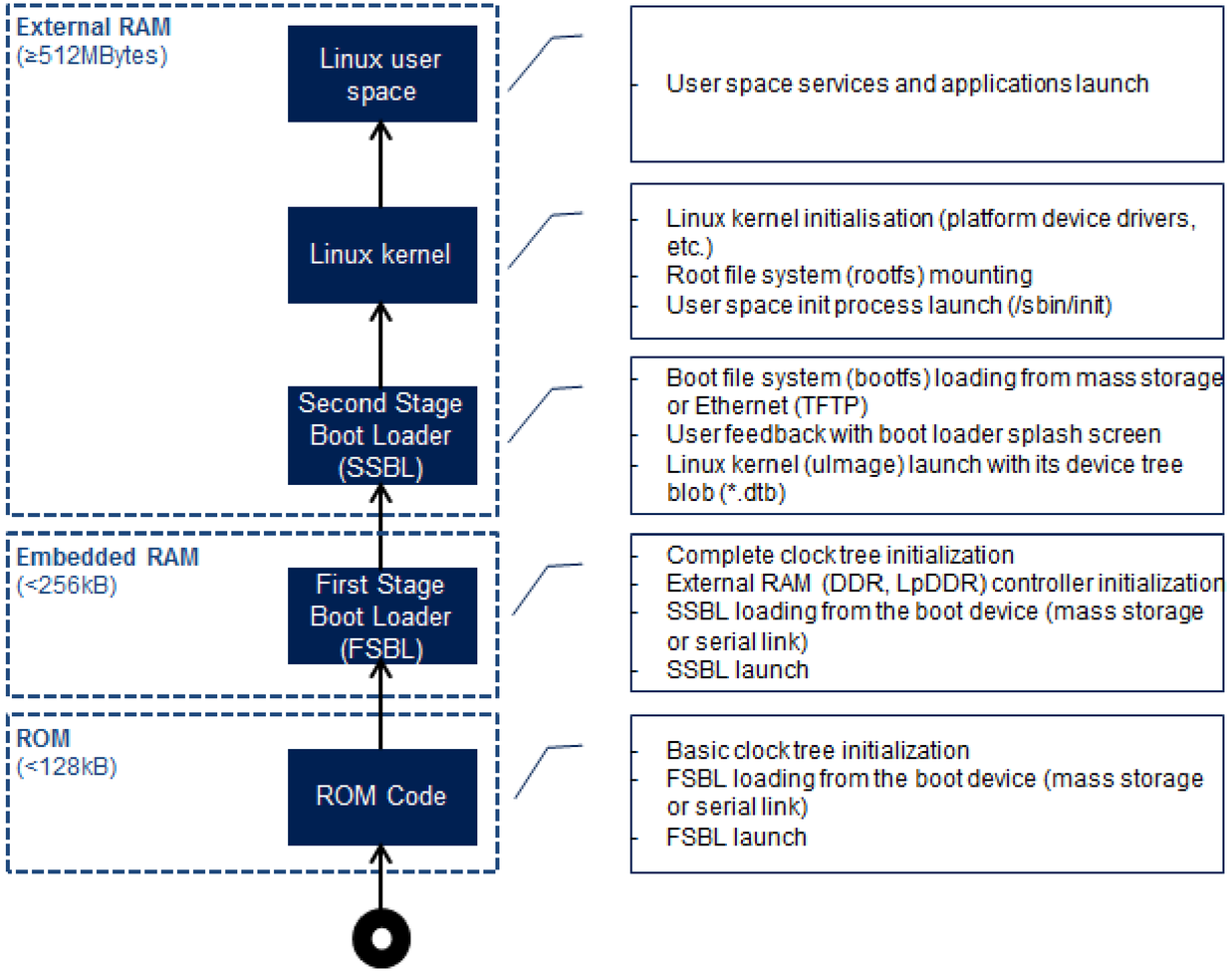
› STM32MP157C-DK2 board forces BOOT1 to 0 by hardware

› It does not have NOR nor QSPI memory

Boot modes	BOOT 2	BOOT 1	BOOT 0
UART/USB	0	0	0
~~NOR/QSPI~~	~~0~~	~~0~~	~~1~~
Reserved	**1**	**0**	**0**
SD-Card	**1**	**0**	**1**

SD-Card mode

- MP is booted first
- The MC is powered up by the MP
- Source clocks are configured by the FSBL
- Two boot types are available :
  - Secure boot (not covered by this presentation)
    - FSBL is TF-A (Trusted Firmware-A)
  - Basic mode
    - FSBL is U-Boot SPL (Secondary Program Loader)



Reserved mode (not so much ;))

- Reserved mode is a debugging mode for the MC (Cortex-M4)
  - Also called Engineering mode
- Only the MC is started
- MP (Cortex-A7) is halted
- All clocks have to be setup by The MC itself
- It has to be re-flashed after each reboot
  - Due to no flash memory onboard

RIOT OS - Engineering mode

Nearly like others STM32

Nearly...

Cortex-M4 SRAM overview

- STM32MP1 family uses 4 SRAM contigous banks at adress 0x10000000 for a total of 384 KB:
  - SRAM1 : 128 KB
  - SRAM2 : 128 KB
  - SRAM3 : 64 KB
  - SRAM4 : 64 KB
- A RETRAM of 64 KB :
  - available and not erased after standby due to an external power supply
  - Starts at address 0x00000000 which is also the default address of the vector table

Cortex-M4 SRAM from RIOT

- RIOT uses VTOR (VecTOR Remap) register from SCB (System Control Block) to redefine vector table address
- But where is the ELF binary loaded as there is no flash memory ?
  - Wherever you decide in the RAM !
  - Because « ROM » is fake and is a part of SRAM/RETRAM.
  - This pseudo « ROM » will be flashed using onboard ST-Link V2 programmer once the target is powered up.
- Thus ROM and RAM size can be customized according to your needs.
- From **cpu/stm32/stm32_mem_lengths.mk** :

```
else ifeq ($(STM32_TYPE), MP)
  ifeq ($(STM32_FAMILY), 1)
    ifeq ($(STM32_MODEL), 157)
      RAM_START_ADDR ?= 0x10000000
      ifdef STM32MP1_ENGINEERING_MODE
        RAM_LEN ?= 320K
      endif
        RAM_LEN ?= 384K
    endif
  endif
endif
```

```
else ifeq ($(STM32_TYPE), MP)
  ifeq ($(STM32_FAMILY), 1)
    ifeq ($(STM32_MODEL), 157)
      ifdef STM32MP1_ENGINEERING_MODE
        ROM_START_ADDR ?= 0x10005000
      else
        ROM_START_ADDR ?= 0x0
      endif
      ROM_LEN ?= 64K
    endif
  endif
else
```

Clocks

- Source clocks to setup :
  - HSI: High Speed Internal clock
  - LSI: Low Speed Internal clock
  - HSE: High Speed External clock
  - LSE: Low Speed External clock
  - CSI: Low Power Internal clock
- Four PLL dedicated to peripherals

```
/* give the target core clock (HCLK) frequency [in Hz],
 * maximum: 209MHz (rounded) */
#define CLOCK_CORECLOCK     MHZ(208)
/* 0: no external high speed crystal available
 * else: actual crystal frequency [in Hz] */
#define CLOCK_HSE           MHZ(24)
/* 0: no external low speed crystal available,
 * 1: external crystal available (always 32.768kHz) */
#define CLOCK_LSE           (1U)
/* 0: no internal oscillator
 * else: actual internal oscillator frequency [in Hz] */
#define CLOCK_HSI           MHZ(64)
```

- STM32 clocks configuration tool :

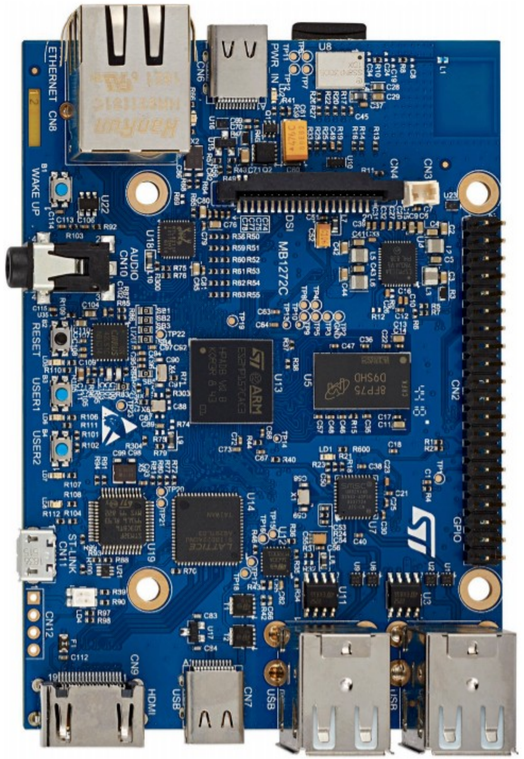
  usage: cpu/stm32/dist/clk_conf/clk_conf <cpu_model> <coreclock> <hse_freq> <lse> [pll_i2s_src] [pll_i2s_q_out] [pll_sai_q_out]

  $ cpu/stm32/dist/clk_conf/clk_conf stm32mp157 208000000 24000000 3276

```
/* Main PLL factors */
#define CLOCK_PLL_M         (2)
#define CLOCK_PLL_N         (52)
#define CLOCK_PLL_P         (3)
#define CLOCK_PLL_Q         (13)
```

Peripherals

- Only few peripherals are supported at this time:
- Timers
- UARTs
- GPIOs
- I2C (not tested yet)



Tools

- Build Linux distro : **Ubuntu 20.04**
  - Install required packages :
    - sudo apt install build-essential gcc-arm-none-eabi gdb-multiarch
- OpenOCD
  - No tag since a long time so use last commit on master at the time of this presentation
    - $ git clone https://git.code.sf.net/p/openocd/code openocd
    - $ cd openocd
    - $ git checkout 393448342 -b mp1_ocd
    - $ make -j$(nproc) && sudo make install

Build and test :)

- Pull request is still in review at this time, thus use custom repository :
  - $ git clone https://github.com/gdoffe/RIOT.git -b mp1_dev
  - $ cd RIOT
  - $ make BOARD=stm32mp157c-dk2 -C tests/periph_gpio/ flash

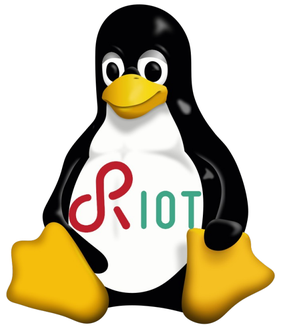
RIOT OS – Did you say Linux ?

Buildroot

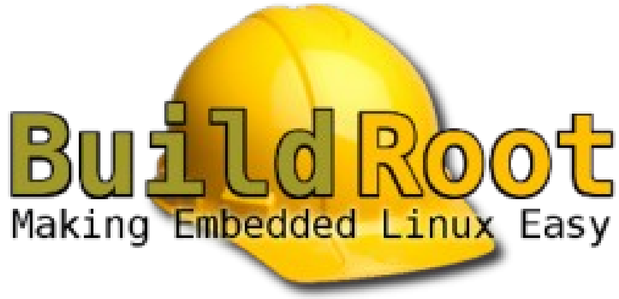
The device tree

remoteproc

A word about Linux

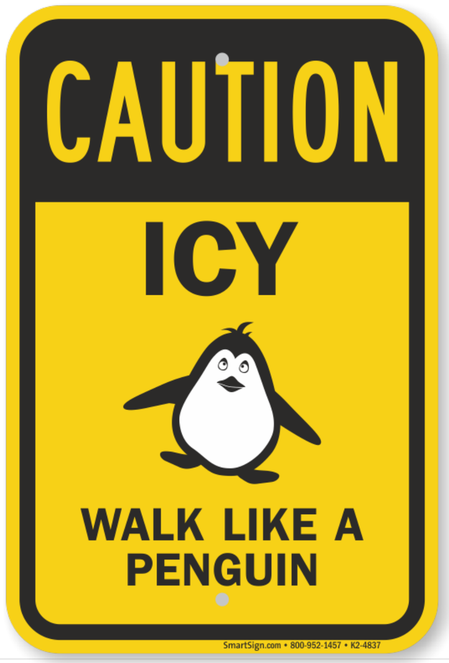


- Linux distribution used : buildroot
  - Linux kernel version at this time : 5.7
  - Excellent tutorials from Bootlin : https://bootlin.com/blog/building-a-linux-system-for-the-stm32mp1-basic-system/
  - Simple and efficient :
    - $ git clone git://git.buildroot.net/buildroot
    - $ cd buildroot
    - $ make stm32mp157c_dk2_defconfig
    - $ make
- Flash buildroot on sdcard :
  - $ sudo dd if=output/images/sdcard.img of=/dev/mmcblk0 bs=1M



The device tree

- The device tree is an universal way to describe hardware for bootloaders and Operating Systems.
- U-boot and Linux kernel use their own device tree
- Clocks (HSE, LSE, HSI, LSI, PLL, etc...) are configured by the FSBL :
  - **NEVER UPDATE SOURCE CLOCKS IN FIRMWARE !!!**




```
> find arch/arm/boot/dts/ -name stm32mp1*
arch/arm/boot/dts/stm32mp15xxab-pinctrl.dtsi
arch/arm/boot/dts/stm32mp157c-ed1.dts
arch/arm/boot/dts/stm32mp157c-dhcom-pdk2.dts
arch/arm/boot/dts/stm32mp157c-dk2.dts
arch/arm/boot/dts/stm32mp157c-ev1.dts
arch/arm/boot/dts/stm32mp15xx-dkx.dtsi
arch/arm/boot/dts/stm32mp157a-avenger96.dts
arch/arm/boot/dts/stm32mp151.dtsi
arch/arm/boot/dts/stm32mp15-pinctrl.dtsi
arch/arm/boot/dts/stm32mp15xxaa-pinctrl.dtsi
arch/arm/boot/dts/stm32mp157.dtsi
arch/arm/boot/dts/stm32mp157a-dk1.dts
arch/arm/boot/dts/stm32mp15xxac-pinctrl.dtsi
arch/arm/boot/dts/stm32mp153.dtsi
arch/arm/boot/dts/stm32mp15xxad-pinctrl.dtsi
arch/arm/boot/dts/stm32mp157c-dhcom-som.dtsi
arch/arm/boot/dts/stm32mp15xc.dtsi
```

remoteproc

- Linux uses the remoteproc kernel framework to load the MC firmware into SRAM and start it.
- As the MP is already started and configured, it is impossible to change VTOR to update the vector table address.
  - Firmware must be loaded in RETRAM (0x00000000).
- Firmwares are stored under /lib/firmware/ directory.
- To set the firmware to load :
  - $ echo rproc-m4-fw > /sys/class/remoteproc/remoteproc0/firmware
- To start/stop the firmware :
  - $ echo start > /sys/class/remoteproc/remoteproc0/state
  - $ echo stop > /sys/class/remoteproc/remoteproc0/state

```
m4_rproc: m4@10000000 {
    compatible = "st,stm32mp1-m4";
    reg = <0x10000000 0x40000>,
          <0x30000000 0x40000>,
          <0x38000000 0x10000>;
    resets = <&rcc MCU_R>;
    st,syscfg-holdboot = <&rcc 0x10C 0x1>;
    st,syscfg-tz = <&rcc 0x000 0x1>;
    status = "disabled";
};
```

```
&m4_rproc {
    memory-region = <&retram>, <&mcuram>, <&mcuram2>, <&vdev0vring0>,
            <&vdev0vring1>, <&vdev0buffer>;
    mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;
    mbox-names = "vq0", "vq1", "shutdown";
    interrupt-parent = <&exti>;
    interrupts = <68 1>;
    status = "okay";
};
```

rproc in U-Boot

- Nearly the same than Linux
- First initialize the rproc framework
  - # rproc init
- Get the firmware to load :
  - # ext4load mmc 0:4 ${kernel_addr_r} /lib/firmware/rproc-m4-fw
- And load it :
  - # rproc load 0 ${kernel_addr_r} ${filesize}
- To start/stop the firmware :
  - # rproc start 0
  - # rproc stop 0

In the future

TODO list

- Finalize the opened pull request : https://github.com/RIOT-OS/RIOT/pull/14691
- Submit all remaining code as a new pull request
- Test support of all STM32 peripherals (QDEC, SPI, PWM, …)
- Implement Hardware SEMaphore (HSEM) to protect shared ressources (EXTI, GPIOs)
- Implement InterProcessor Communication protocol (IPCC) => (core/mbox ?)
- Linux Yocto/Buildroot packaging (thanks to KConfig)

Thank you!