STM32CubeIDE+MX part 2

Interrupts and events

- … are both mechanisms used to handle asynchronous occurrences, but
  - an interrupt can do two things:
    - cause software to execute via an interrupt handler, and
    - cause some peripheral to do something
  - an event can cause some peripheral to do something
- Interrupts are immediate signals that temporarily halt the processor to handle urgent tasks, while events are notifications that can be processed sooner or later
- *If you need "to execute a few lines of code", you need to use interrupts*
- With events there is no ISR (Interrupt Service Routine), so
  - lower latency and overhead
- We shall see an example of event later on

Interrupts

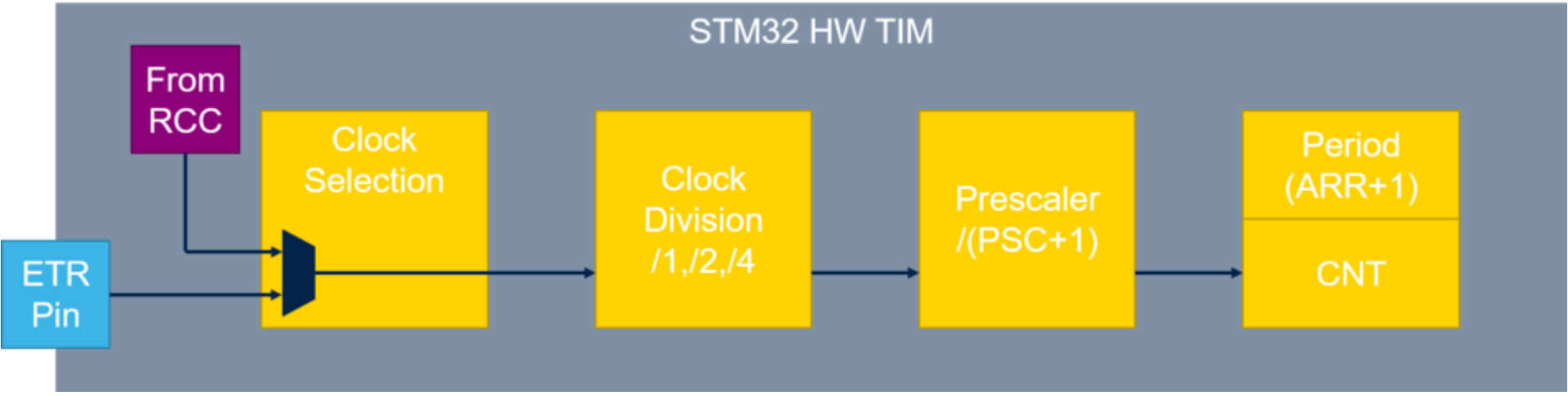
- The blue button (B1) is connected to PC13
- In MX
  - we set PC13 to GPIO_EXTI13
  - in GPIO: External Interrupt… with rising edge…
    - pushing B1 puts the line high
  - and enable EXTI line [15:10] in NVIC
- In IDE, test it with *interrupt*
  - interrupt -> *EXTI15_10_IRQHandler* (in *stm32h7xx_it.c*)
  - -> *HAL_GPIO_EXTI_IRQHandler* (in *stm32h7xx_hal_gpio.c*, which is in *Drivers…STM32H7xx_HAL_Driver… Src*)
  - -> *HAL_GPIO_EXTI_Callback*
- You may want to try
  - External… with falling edge…

Interrupt priorities

- The blue button (B1) is connected to PC13
- Connect a button (with pull-down resistor) to PB8
- In MX
  - GPIO:
    - set PC13 as before
    - set PB8 to GPIO_EXTI8
    - and enable the interrupt EXTI line [9:5] in NVIC
  - NVIC: set two different priorities for the two EXTI lines
- IDE:
  - these commands appear at the end of *main.c*
    - HAL_NVIC_SetPriority(…); HAL_NVIC_EnableIRQ(…);
  - test *int_priorities*

Timer

- The MCU has a low-power real-time clock (RTC) and several general-purpose 16-bit timers, PWM timers for motor control, low-power timers.
- Timer TIM1 is an advanced-control timer which can be used for [RM0455, p. 1447]
  - Input capture
  - Output compare
  - PWM generation (edge- or center-aligned modes)
  - One-pulse mode output



Timer: time interval counter

- In MX:
  - Timers… TIM1…
  - Clock Source: Internal Clock
  - Parameter Settings:
    - PSC=9599
  - so that, for counter increment rate:
    - TIM1 is on the APB2 bus: 96 MHz (*please verify*)
    - 96 MHz / (9599+1) = 10 kHz

Timer: time interval counter

- In IDE:
  - Project… Properties… C/C++ Build… Settings… Tool Settings… MCU/MPU Settings: enable “Use float with printf…”
  - then test *TIM1*

Timer generated interrupts

- In MX:
  - Timers… TIM1…
  - Clock Source: Internal Clock
  - Parameter Settings:
    - PSC=11999,
    - Counter Period (ARR) 3999,
    - NVIC: TIM1 update interrupts
  - so that, for counter period:
    - TIM1 is on the APB2 bus: 96 MHz
    - 96 MHz / (23999+1) / (3999+1) = 1 Hz
- In IDE: test *TIM1int*

PWM

Example *PWM*

- In MX:
  - TIM1,
  - Clock Source: Internal Clock,
  - Channel4: PWM Generation CH4
  - GPIO: PE14 [DS13195 p. 72; PA11 would also be possible]
- In IDE: connect a led to PE14, and test *PWM*

Example *servo*

- a servo needs *f* = 50 Hz, *D* = 5-10%
- PSC=1919, ARR=999 –> *f*=50 Hz
- connections:
  - 5 V on red wire (not 3.3 V), GND on black/brown wire
  - PWM signal on yellow wire

RTC

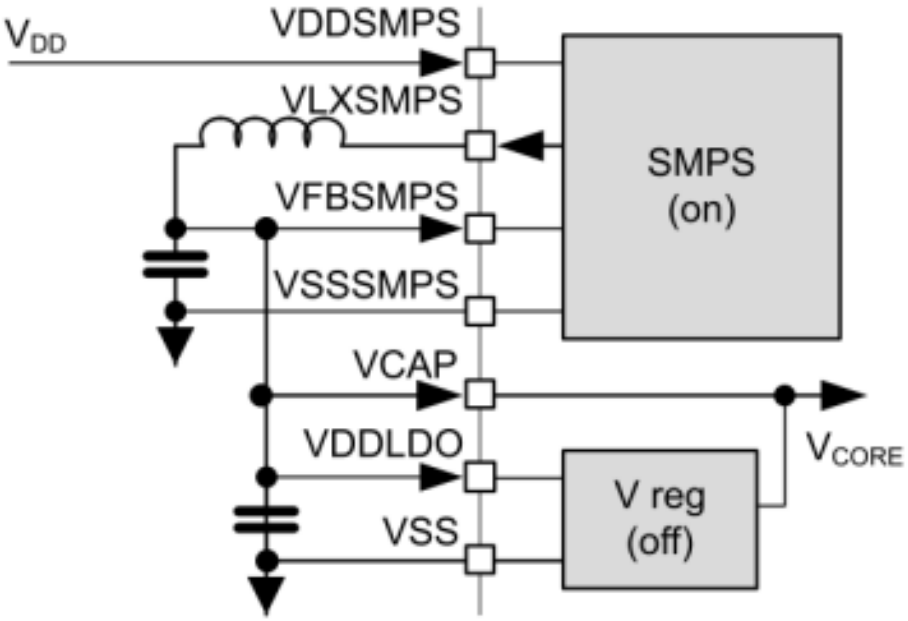
- A Real Time Clock is an independent timer/counter
  - calendar with subseconds, seconds, minutes, hours, day, date of day in month, month and year
    - automatic correction for leap-year
  - daylight saving compensation programmable by software
  - programmable alarm with interrupt function
  - automatic wakeup to manage all low-power modes
- The RTC clock sources can be, among others:
  - a 32.768 kHz external crystal (LSE)
  - the internal low-power RC oscillator (LSI, with typical frequency of 32 kHz)
- The RTC clock is often also powered by a (small) battery, in order to keep date/time even without main power

RTC [UM2217, p. 3480]

- In MX: Timers… RTC…
  - Activate Clock Source and Calendar
  - Parameter Settings: set current date and time
- In IDE: test *RTC*

Power supply [UM2408, p. 18]

- Power supply can be provided by five different sources
  - default: host PC connected to CN1 through USB cable
- LDO (low dropout regulator) then reduces 5 V to 3.3 V
- VDD is set to 3.3 V by JP5
- Different internal SMPS / LDO (switched mode power supply) configurations are supported
  - default: internal SMPS only
- VCORE ~ 1.0-1.3 V



2. Direct SMPS supply

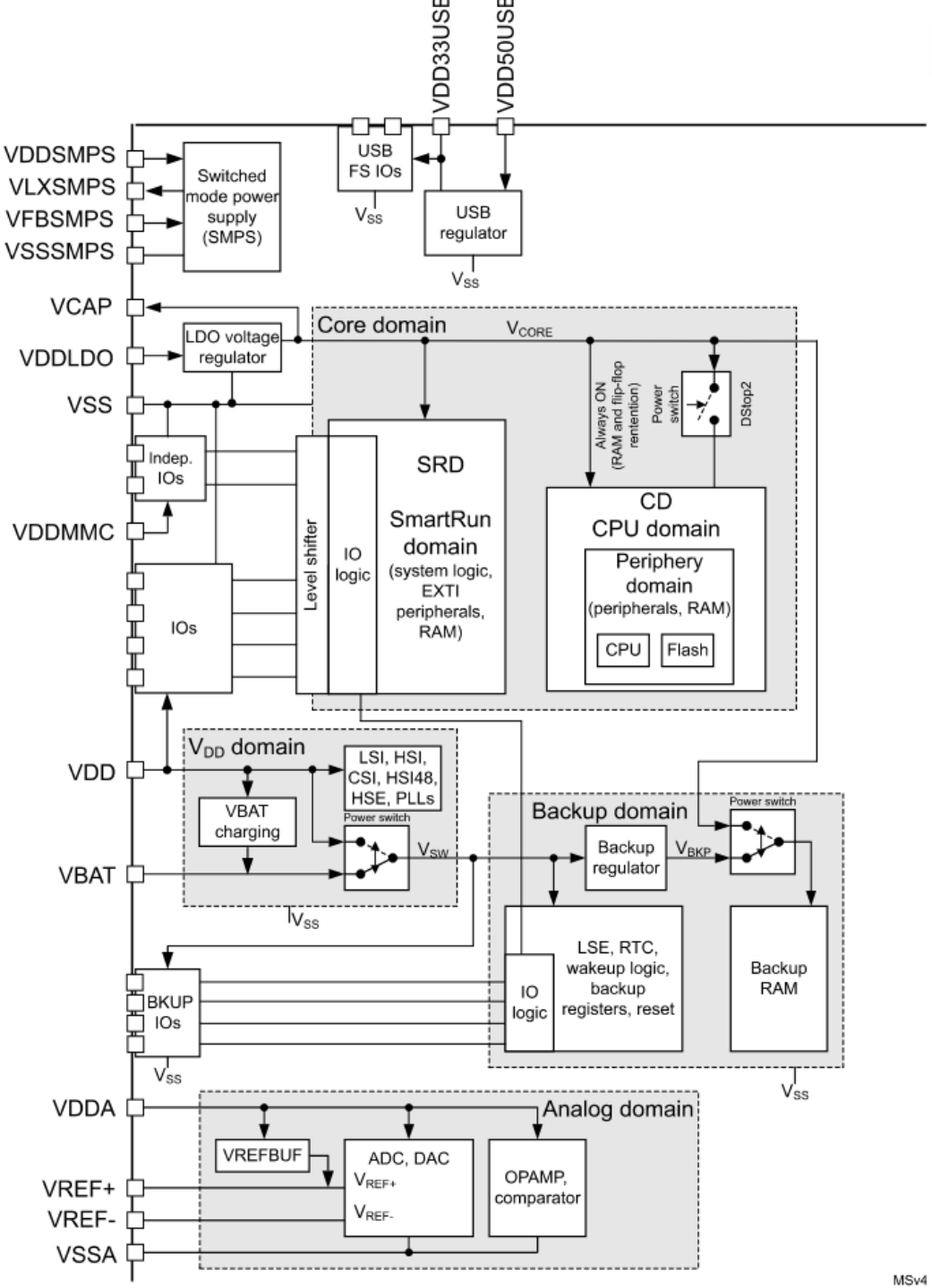
Low-power modes [DS13195, p. 16, RM0455, p 290]

- There are several ways to reduce power consumption on STM32H7A3xI/G:
  - decrease dynamic power consumption by
    - slowing down the system clocks,
    - reducing voltage (*voltage scaling*),
    - individually clock gating the peripherals that are not used
  - save power consumption when the CPU is idle, by selecting among the available low-power modes according to the user application needs.

Power domains [RM0455, p. 260]

- Power supply is divided into several domains
  - Core domains (VCORE)
    - CPU domain (CD) containing most peripherals and the Cortex®-M7 Core (CPU)
    - SmartRun domain (SRD) containing low-power peripherals and system control
  - VDD domain, for I/O, power management, oscillators
  - External VDD_MMC I/O domain, for some I/O
  - Backup domain (VSW, VBKP), for RTC, backup registers, backup RAM
  - Analog domain (VDDA), to reduce noise in analog part

Power supply overview [RM0455, p. 265]



I/O voltage is VDD

- MX: enable PB1 as output
- Connect a voltmeter to PB1
- IDE: test *18-33_V.c* and change VDD using JP5

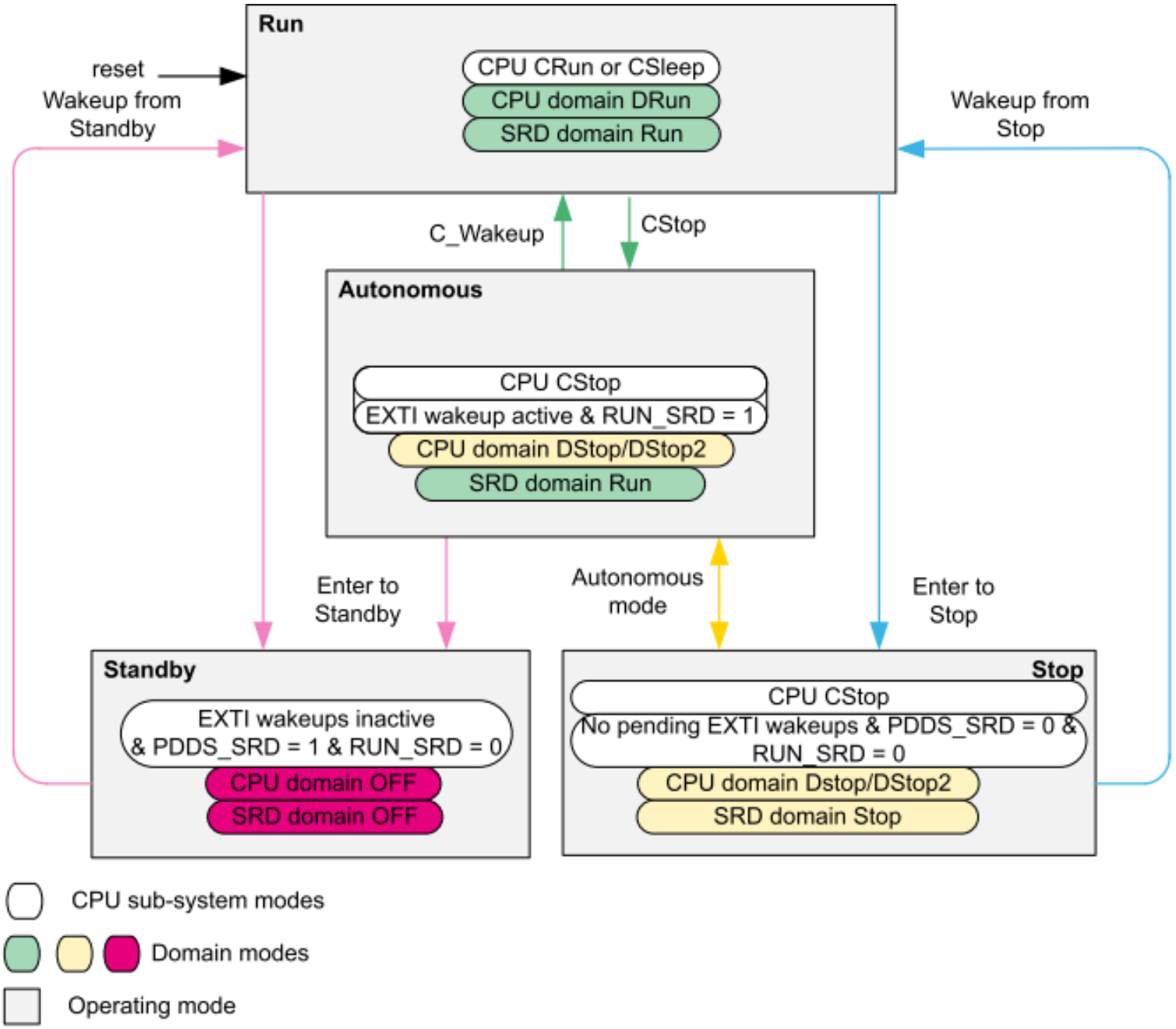
System operating modes [RM0455 p. 287]

- Several system operating modes are available to tune the system according to required performance
- The operating modes allow
  - controlling the clock distribution to the different system blocks
  - and powering them.
- Except for Standby, several voltage scaling can be selected

Operating mode summary [RM0455 p. 289]

System	SRD domain	CPU domain	CPU	Entry	Wakeup	System oscillator	System clock	Domain bus matrix clock	Peripheral clock	CPU clock	Voltage regulator	Domain supply	Voltage scaling
Run	SRDRun	DRun	CRun	-	-	ON	ON	ON	ON	ON	ON	ON	VOS0/1/2/3
			CSleep	WFI or return from ISR or WFE	Any interrupt or event				ON/OFF[1]	OFF			
Autono-mous		DStop[2]	CStop	SLEEPDEEP bit + WFI or return from ISR or WFE	Any EXTI interrupt or event			OFF	ON/OFF[3]			RET[9]	
		DStop2[4]											
Stop[5]	SRDStop	DStop[2]		SLEEPDEEP bit + WFI or return from ISR or WFE or wakeup source cleared[6]		ON/OFF[7]	OFF		OFF[8]			ON	SVOS3/4/5
		DStop2[4]										RET[9]	
Standby[10]				PDDS_SRD bit + SLEEPDEEP bit + WFI or return from ISR or WFE or wakeup source cleared[6]	WKUP pins rising or falling edge, RTC alarm (Alarm A or Alarm B), RTC wakeup event, RTC tamper events, RTC timestamp event, external reset in NRST pin, IWDG reset	OFF			OFF		OFF	OFF	OFF

Power-control modes state diagram [RM0455 p. 292]



Main low-power modes [UM2217, p. 1212]

- 3 main low-power modes:
  - SLEEP mode : Cortex-Mx is stopped and all PWR domains are remaining active (powered and clocked)
  - STOP mode : Cortex-Mx is stopped, clocks are stopped and the regulator is running
  - STANDBY mode : All PWR domains enter DSTANDBY mode and the VCORE supply regulator is powered off
    - when exiting STANDBY, the MCU *restarts*, like after a reset!

Examples of low-power modes

- SLEEP and STOP mode
  - In MX, set the interrupt EXTI13 for the blue button
  - In the IDE, test *lowpower* and measure the current at JP4
    - WFI: Wait For Interrupt
- SLEEP mode with timer TIM1
  - In MX, set the interrupt for TIM1
  - In the IDE, test TIM1*int_sleep*

- SLEEP and STOP mode *with WFE (Wait For Event)*
  - In MX, set PC13 (the blue button) as
    - GPIO_EXTI13
    - GPIO Mode: External Event Mode with Rising Edge
  - In the IDE, test *PC13_event*

RTC and wakeup

- In MX: Timers… RTC…
  - Activate Clock Source, Wake Up Internal
  - Parameter Settings:
    - Wake UP, Clock 1 Hz, Counter 4
    - NVIC Settings: Enable
- In IDE
  - for STOPMode, test *RTCwakeup*
  - for STANDBYMode, test *RTCwakeup2*

Using a slower clock

- Default clock speed is 96 MHz
  - obtained from HSE at 8 MHz (STLK_MCO/PA8)
    - generated by the other MCU from a 25 MHz osc. (X1)
- In MX: Clock Configuration… CDCPRE: set to /16
  - all the clocks will be at 6 MHz
- In the IDE,
  - in the middle of *main.c*, two lines have changed

```
RCC_ClkInitStruct.SYSCLKDivider =
        RCC_SYSCLK_DIV16;
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct,
        FLASH_LATENCY_0) != HAL_OK)
```

  - test again *blinky*
    - current decreases from 14.8 to 4.4 mA

Watchdog (WDG)

- The main goal is to detect and resolve malfunctions due to software failures.
  - e.g., to stay stuck in a particular stage of processing
- S/w has to periodically refresh the WDG counter (*pet the dog*)
  - if the counter isn't refreshed, a system reset is generated (*the dog barks*)
- There are two types of WDG:
  - WWDG : Window watchdog
    - clocked from the main clock; it has an early warning interrupt capability (Early Wakeup Interrupt, EWI)
  - IWDG : Independent watchdog
    - clocked from an independent low-speed internal (LSI) 32 kHz internal RC
    - it can operate in Stop and Standby mode

Watchdog

- In MX:
  - System Core… IWDG1: activated
    - Prescaler: 128
    - Window value: 1000
    - Reload value: 1000
  - so that f = 32 kHz / (128*1000) = 0.25 Hz, i.e. T = 4 s
- In IDE: test *IWDG*

Power supply supervisor [DS13195 p. 15, RM0455 p. 281]

- *Power-down reset* (PDR): monitors V_{DD} power supply
  - a reset is generated when V_{DD} drops below a fixed threshold
- *Brownout reset* (BOR): monitors V_{DD} power supply
  - 3 BOR thresholds (from 2.1 to 2.7 V) can be configured
  - a reset is generated when V_{DD} drops below the selected threshold
- *Programmable voltage detector* (PVD) monitors the V_{DD} power supply by comparing it with a threshold selected from a set of predefined values
- V_{BAT} threshold
  - VBAT battery voltage level can be monitored by comparing it with 2 thresholds levels

Power PVD

- In MX:
  - Power… PWR…
  - Power Voltage Detector: … In (Internal…)
  - Parameter Settings…
    - PVD det. Level: 2.85 V
    - PWR PVD mode: Ext. Int. Mode with Falling edge…
  - Enable the PVD interrupt in NVIC
- Test *PWR-PVD.c,* using a potentiometer to reduce V_{DD}

Temperature monitoring [DS13195 p. 15, RM0455 p. 281]

- Temperature threshold
  - a dedicated temperature sensor monitors junction temperature and compare it with 2 threshold levels

V_{BAT} operation [DS13195 p. 21, RM0455 p. 281]

- V_{BAT} power domain contains RTC, backup registers and backup SRAM
- To optimize battery duration, it is supplied by
  - V_{DD} when available, or
  - the voltage applied on V_{BAT} pin
- V_{BAT} power is switched on when PDR detects that V_{DD} dropped below the PDR level
- Voltage on V_{BAT} pin can be provided by
  - external battery or supercapacitor, or
  - directly by V_{DD} (in which case, the V_{DD} mode is not functional)
- An internal V_{BAT} battery charging circuitry can be activated when V_{DD} is present

Tutorial and examples

… are available here

- https://wiki.st.com/stm32mcu/wiki/Microcontroller
- https://wiki.st.com/stm32mcu/wiki/Category:Getting_started_with_STM32_system_peripherals
- https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-online-training/stm32h7-online-training.html

More exercises / homework

- UART to have two boards talking each other
- Several digital I/O pins to write to the 16x2 display 162B
- Humidity/temperature sensor
- FIR filter
  - or (it’s the same…) a SISO control system
- Signal generation

Possible projects

- Attitude control of a quadrotor drone
- Musical keyboard control and sound generation

LL drivers

- Low-layer APIs (LL) offer fast, light-weight, expert-oriented layer which is *closer to hardware than HALs*
- LL services:
  - do not require any additional memory resources to save their states, counter or data pointers
  - operations are performed by changing content of associated peripheral registers
- Unlike HALs, LL APIs are not provided for peripherals for which
  - optimized access is not a key feature, or
  - heavy software configuration is required (e.g., USB)

LL drivers

- HAL and LL are complementary
  - HAL offers high-level and feature-oriented APIs with a high-portability level; they hide MCU and peripheral complexity
  - LL offers low-level APIs at register level, with better optimization but less portability; they require deep knowledge of the MCU and peripheral specifications
- It is possible to use both HAL and LL drivers
  - one can handle the IP initialization phase with HAL and then manage the I/O operations with LL drivers
- Major difference between HAL and LL:
  - HAL drivers require handles for operation management, while
  - LL drivers operates directly on peripheral registers

LL drivers

- In CubeMX
  - Project Manager… Advanced Settings:
    - GPIO: set LL instead of HAL
- In CubeIDE, in main.c:
  - LL_GPIO_SetOutputPin(GPIOB, LL_GPIO_PIN_0);
  - LL_mDelay(100);
  - LL_GPIO_ResetOutputPin(GPIOB, LL_GPIO_PIN_0);
  - LL_mDelay(100);