UNIVERSITÉ SAVOIE MONT BLANC

Low Power modes on STM32

Lecture version 1

June 2021

Sylvain MONTAGNY

www.linkedin.com/in/sylvainmontagny

sylvain.montagny@univ-smb.fr

Synopsis

This PDF book is distributed free of charge on our website and can be used for any purposes. Any suggestions are welcome and can be proposed to the Author. This academic content is part of the Electronics and Embedded Systems Master degree at "Savoie Mont-Blanc" University (France).

The lecture explains in detail the main low power modes of the STM32 microcontrollers (L0 and F4 series). It also provides tips to drastically reduce power consumption when engineers develop firmware for Cortex M.

Source

The content of the book comes from a compilation of various documentations, datasheets, reference manual, application notes from ST. Some examples and explanations come from the excellent Fastbitlab STM32 lecture.

Related document

This document is part of a set of resources on IoT and LPWAN (**L**ow **P**ower **W**ide **A**rea **N**etwork).

- A free PDF Book on LoRa-LoRaWAN
- 130 short videos on LoRa LoRaWAN and IOT
- Two days training with online instructor

Table of contents

1 Materials and documentations

1.1 MCU and Nucleo board

This book provide many examples tested on two Nucleo boards but all Nucleo Board should work.

We are going to use two different microcontrollers on Nucleo boards: the STM32F446RE and the STM32L073RZ. STM32L073 is aimed for low power application while STM32F446 is used for small DSP applications.

Note: For some reasons, the power consumption between two Nucleo boards with the same MCU are slightly different from one another. Therefore, you will have to keep the same Nucleo STM32F446 and the same Nucleo STM32L073 during all your experiments.

1.1.1 The STM32F446RE

Features:

- Cortex®-M4 CPU with FPU, ART Accelerator™, frequency up to 180 MHz, DSP instructions
- 225 DMIPS/ 1.25 DMIPS per MHz (Dhrystone 2.1)
- Memories: 512 kB of Flash memory - 128 KB of SRAM
- Low power - Sleep, Stop and Standby modes – VBAT supply for RTC

Links for documentation:

- Reference Manual STM32F446xx : RM0390 from ST
- Datasheet STM32F446xC/E from ST
- Cortex M4 Device : Generic User Guide from ARM

1.1.2 The STM32L073RZ

Features:

- ARM® 32-bit Cortex®-M0+ with MPU - From 32 kHz up to 32 MHz max
- 0.95 DMIPS per MHz
- Memories: 192 KB Flash memory - 20KB RAM - 6 KB of data EEPROM
- 0.29 µA Standby mode - 0.43 µA Stop mode (16 wakeup lines)

Links for documentation:

- Reference Manual STM32L0x3 : RM0367 from ST
- Datasheet STM32L073xZ from ST
- Cortex M0 Device : Generic User Guide from ARM

1.2 The X-NUCLEO-LPM01A measurement board

We are using the X-NUCLEO-LPM01A board to measure de current/energy consumption. This board is working with voltage from **1,8V to 3,3V**. It measures dynamic current **up to 50 mA** with a maximum **100 khz bandwith**.

Link for documentation:

- X-NUCLEO-LPM01A expansion board User manual : UM2243 from ST

We need an micro USB cable to power this board.

1.2.1 Power supply Overview of the Nucleo board

On the Nucleo board, Jumper 5 (JP5) controls several possibilities to power the entire Nucleo board.

1. USB 5V (U5V) : From the usual USB mini connector (CN1).
2. E5V (external power supply 5V): From the Morpho connector (CN7 pin 6).
3. Vin (external power supply from 7 to 12V): From the Arduino connector (CN6 pin 8).

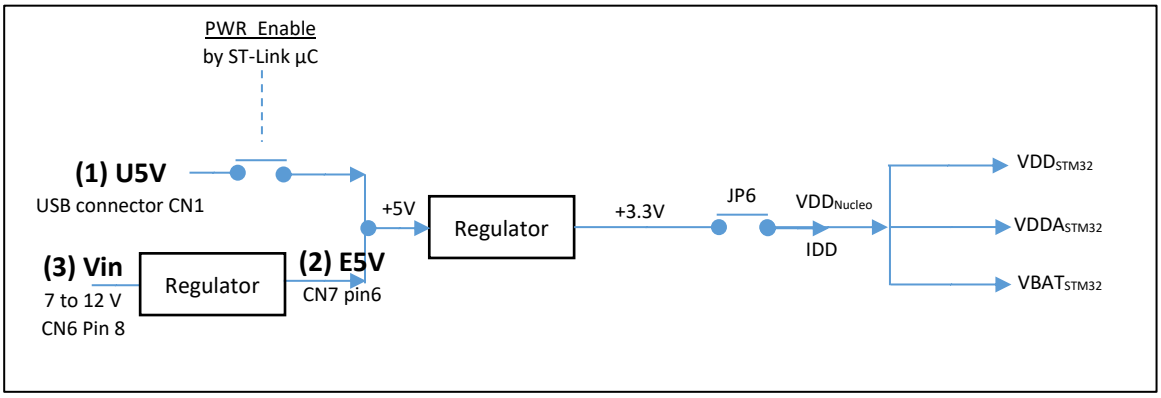


Figure 1 : Power supply overview for the Nucleo board

We want to measure the current IDD, which represents the overall current consumed by the µC. On the Nucleo board, this IDD current goes through JP6, than powers the MCU via:

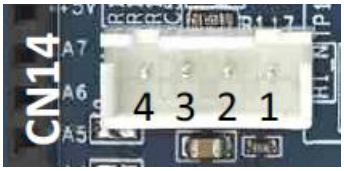
- VDD_{STM32} (VDD domain)
- $VDDA_{STM32}$ (Analog domain) thanks to SB45 (**S**oldier **B**ridge 45)
- $VBAT_{STM32}$ (Backup domain) thanks to SB57 (**S**oldier **B**ridge 57)

Therefore, a current measurement on JP6 provides the overall current consumption of the MCU.

1.2.2 Measurement via the white CN14 Connector of X-NUCLEO-LPM01A

> **This will be the preferred way to measure the power consumption during all labs.**

The CN14 of the X-NUCLEO-LPM01A provide four signals. Only **Pin 3** and **Pin 1** are useful here.



CN14 pin	Signal	Usage
Pin 1	**GND** (-)	Ground of the target
Pin 2	VDD	Alternate power supply source (not measured)
Pin 3	**VOUT** (+)	Positive connection of the target, current is measured
Pin 4	VOUT_MONITORING	Mirror of VOUT. Allows VOUT monitoring without impacting current/power measurements

Table 1: PIN connections between the Nucleo and the LPM01A module

- The ST Link and all the other components of the Nucleo board will be powered by the U5V coming from the USB.
- STM32 will be powered by the X-NUCLEO-LPM01A board.

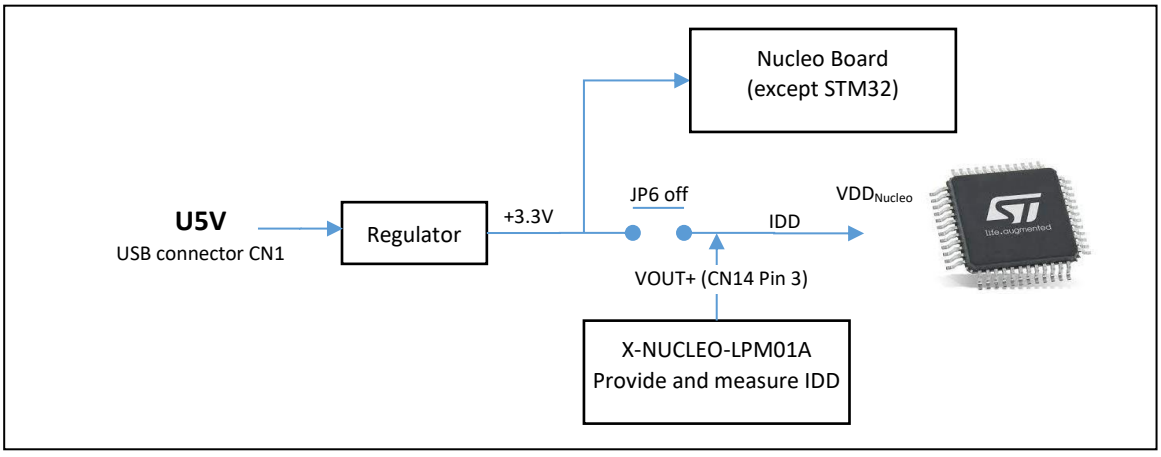


Figure 2: Power and measurement of the Nucleo board and STM32

The X-NUCLEO-LPM01A will provide VDD and measure the IDD current. Here are the configurations and connexions of both Nucleo and X-NUCLEO-LPM01A board:

Nucleo board:

- Remove JP6 (IDD)
- Check : JP5 on "U5V", ST-Link On

X-NUCLEO-LPM01A: We follow the instruction of the documentation for JP9, JP10 and JP4

Power output on	Output connectors and pins	JP9 AREF_ARD jumper	JP10 3V3_ARD jumper	JP4 Additional decoupling capacitor jumper
Basic connector CN14	**Vout**: CN14 pin3 **GND**: CN14 pin1	open	open	**Open**: no decoupling capacitor
				Closed: 2.2 µF decoupling capacitor

Figure 3: Configuration of the X-NUCLEO-LPM01A board.

- JP3 (Power Sel) on "USB"
- JP4 (Decoupl) ON
- JP1 on "Normal"
- Remove JP9
- Remove JP10

Connexions between CN14 and the Nucleo Board:

The Table 1 gives the wiring between the X-NUCLEO-LPM01A and the Nucleo board.

X-NUCLEO-LPM01A	Nucleo board
CN14 PIN 3 (VOUT +)	Left Pin on JP6
CN14 PIN 1 (GND)	GND (optional)

Table 1: PIN connections between the Nucleo and the LPM01A module

1.2.3 Measurement via the Arduino Uno Connector

We can use the Arduino connector to plug the X-NUCLEO-LPM01A over the Nucleo board.

We first remind the power scheme of the Nucleo and STM32.

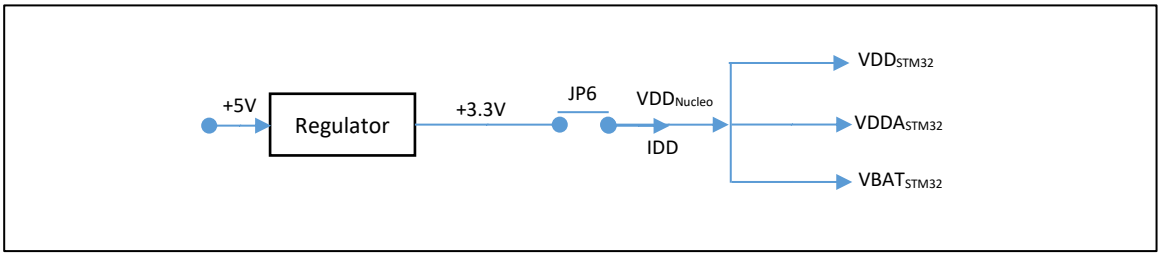


Figure 4: Power of the STM32

Once again, we will have to take JP6 off and power the STM32 with the X-NUCLEO-LPM01A board by using either VDD_{Nucleo}, VDD_{STM32}, $VDDA_{STM32}$ or $VBAT_{STM32}$ because there all connected together. But only $VDDA_{STM32}$ is present on the Arduino connector (CN3 Pin 8), so, if we want to use the X-NUCLEO-LPM01A Arduino connector, we have to provide the power supply through $VDDA_{STM32}$. Be careful not to be confused with all the labels because CN3 pin 8 is reported as AVDD on the Nucleo board, and AREF on the X-NUCLEO-LPM01A.

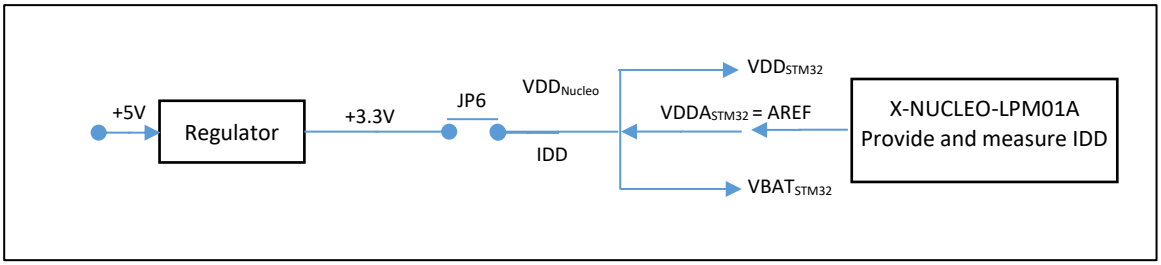


Figure 5: Power and measurement of the Nucleo board and STM32

Here are the configurations and connexions between the Nucleo and the X-NUCLEO-LPM01A board:

Nucleo board:

- Remove JP6 (IDD)
- Check: JP5 on "U5V", ST-Link On

X-NUCLEO-LPM01A: We follow the instruction of the documentation for JP9, JP10 and JP4

Power output on	Output connectors and pins	JP9 AREF_ARD jumper	JP10 3V3_ARD jumper	JP4 Additional decoupling capacitor jumper
Arduino Uno connector, pin AREF	**AREF:** CN3 pin8 **GND:** CN4 pin6 or pin7	closed	open	**Open**: no decoupling capacitor
				Closed: 2.2 µF decoupling capacitor

Figure 6: Configuration of the X-NUCLEO-LPM01A board

- JP3 (Power Sel) on "USB"
- JP4 (Decoupl) ON
- JP1 on "Normal"
- JP9 ON (to put the Power on the AREF/AVDD/VDDA$_{STM32}$ pin)
- Remove JP10

1.3 STM32CubeMonitor-Power

STM32CubeMonitor-Power is the graphical tool for displaying the result of the measurement. You can download the latest version from ST Website along with the USB COM port driver:

- STM32CubeMonitorPower: www.st.com/en/development-tools/stm32cubemonpwr.html
- USB COM port driver: www.st.com/en/development-tools/stsw-stm32102.html

We need to run the measurement once before programming the STM32, otherwise the power is not provided to the MCU (via the CN14 connector and the Arduino connector) and the ST-Link can't write it.

1.4 Debugging issues with low power

Entering low power mode is not straightforward for the debugging operation. The ST-Link connection is easily lost. To prevent this, it's often better to chose the Run mode , instead of the Debug mode .

1.4.1 Debugging with low power modes

If you need to debug your application, you have to configuration some bits depending on the low power mode you want to enter:

```
void HAL_DBGMCU_EnableDBGSleepMode(void);
void HAL_DBGMCU_DisableDBGSleepMode(void);
void HAL_DBGMCU_EnableDBGStopMode(void);
void HAL_DBGMCU_DisableDBGStopMode(void);
void HAL_DBGMCU_EnableDBGStandbyMode(void);
void HAL_DBGMCU_DisableDBGStandbyMode(void);
```

These functions have an influence on power consumption because it keeps the debug capabilities of the MCU. Therefore, we must keep it in mind while measuring low and accurate current.

1.4.2 Fixing the ST Link

If the ST-Link seems not working anymore there are 3 options to resume the debug session:

1. Press the Reset Button while launching a debug session and release it when the ST-Link seems to have overtaken the MCU.
2. Create a simple project (led blink without sleep mode) and generate the .bin executable file (**Project properties > C/C++ Build > Settings > Tool Settings > MCU Post Build Output > Check "Convert to binary files"**). You can program your MCU by dragging the .bin file in the drive which opens in you file system when you plug your Nucleo board.
3. Use ST-Link utility to flash your MCU.

2 Processor modes

2.1 Running Mode

In this first section, we will measure the current consumption with the default project value in STM32CubeMX. We measure the current when the microcontroller is executing instructions and when it has its peripherals enable.

- With STM32CubeIDE default value, create the following "hello world" program which toggles the User Led (PA5) on the Nucleo board, then measure the current consumption during 10 seconds.

Function	Code

main( )	`while (1) {`
	`  HAL_Delay(1000);`
	`  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);`
	`}`

You should approximately find the following values. On each cell, you can write you own result for comparison.

Test Conditions	Current measurement STM32F446	Current measurement STM32L073
Default Mode (CubeMX) Led OFF	17 000 µA	625 µA
Default Mode (CubeMX) Led ON	20 500 µA	3 125 µA

Table 2 : Initial current consumption on STM32F446 and STM32L073 in running mode

What do we see?

We obviously see that the power consumption depends on the LED state. Indeed, the microcontroller's GPIOA Pin 5 powers the User Led. Here, we don't use any low power mode, which means that during the HAL_Delay(1000), the microcontroller resumes its execution.

2.2 Low Power Modes

We will now measure the current consumption while the microcontroller goes in Low Power mode. There are two main "Low power" modes in ARM Cortex M microcontroller.

- Normal Sleep Mode
- Deep Sleep Mode

These Low Power modes (Normal and Deep sleep) are defined by ARM but are often specifically extended by the manufacturer, which is ST in our case. Obviously, STM32L have more low power capabilities than STM32F.

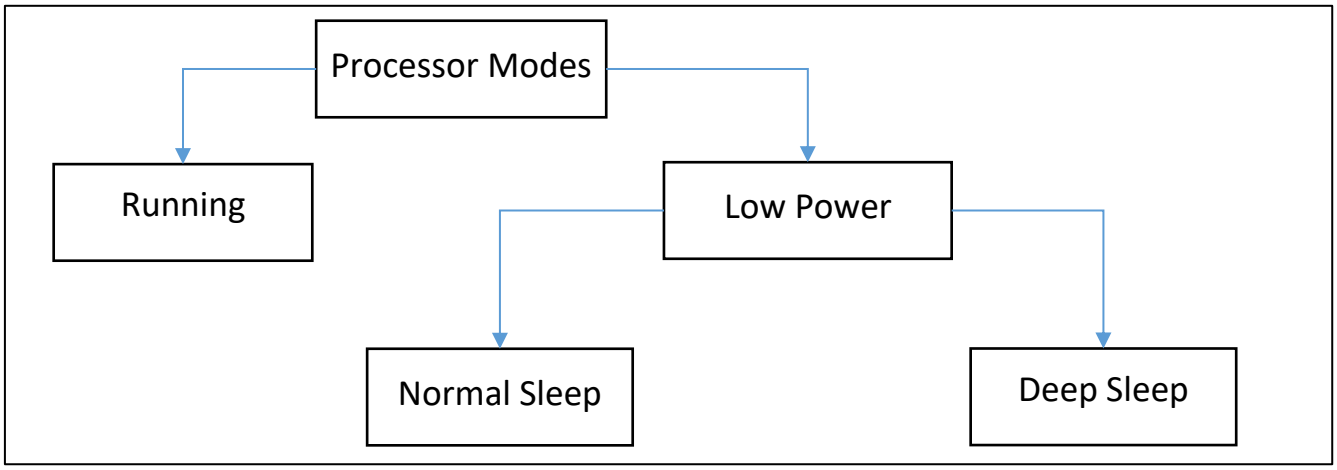


Figure 7: Running and Low Power modes defined by ARM

The choice between the "Normal Sleep" mode and the "Deep Sleep" mode depends on the SLEEPDEEP bit (ARM Generic User Guide). The SLEEPDEEP bit is part of the SCR (System Control Register).

System Control Register

The SCR controls features of entry to and exit from low power state. See the register summary in Table 4-10 on page 4-11 for its attributes. The bit assignments are:

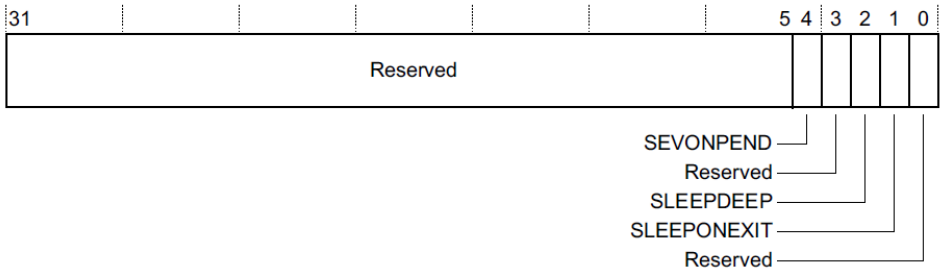


Figure 8: The System Control Register of ARM Cortex M0/M4 processors

For example, our STM32F446 has three low power modes (see Figure 9):

- Normal Sleep: Sleep
- Deep Sleep: Stop and Standby

Mode name	Entry	Wakeup	Effect on 1.2 V domain clocks	Effect on V_{DD} domain clocks	Voltage regulator
Sleep (Sleep now or Sleep-on-exit)	WFI or Return from ISR	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources	None	ON
	WFE	Wakeup event			
Stop	PDDS and LPDS bits + SLEEPDEEP bit + WFI or Return from ISR or WFE	Any EXTI line (configured in the EXTI registers, internal and external lines)	All 1.2 V domain clocks OFF	HSI and HSE oscillators OFF	ON or in low- power mode (depends on *PWR power control register (PWR_CR)*
Standby	PDDS bit + SLEEPDEEP bit + WFI or Return from ISR or WFE	WKUP pin rising edge, RTC alarm (Alarm A or Alarm B), RTC Wakeup event, RTC tamper events, RTC time stamp event, external reset in NRST pin, IWDG reset			OFF

Figure 9: Summary of low power modes in a STM32F446 - Reference Manual

The STM32L073 has five low power modes (see Figure 10)

- Normal Sleep: Low-power run, Sleep and Low-power sleep
- Deep Sleep: Stop and Standby.

Mode name	Entry	Wakeup	Effect on V_{CORE} domain clocks	Effect on V_{DD} domain clocks	Voltage regulator
Low-power run	LPSDSR and LPRUN bits + Clock setting	The regulator is forced in Main regulator (1.8 V)	None	None	In low-power mode
Sleep (Sleep now or Sleep-on-exit)	WFI or Return from ISR	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources	None	ON
	WFE	Wakeup event			
Low-power sleep (Sleep now or Sleep-on-exit)	LPSDSR bits + WFI or Return from ISR	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources, Flash CLK OFF	None	In low-power mode
	LPSDSR bits + WFE	Wakeup event			
Stop	PDDS, LPSDSR bits + SLEEPDEEP bit + WFI, Return from ISR or WFE	Any EXTI line (configured in the EXTI registers, internal and external lines)	All V_{CORE} domain clocks OFF	HSI16(1), HSE and MSI oscillators OFF	In low-power mode
Standby	PDDS bit + SLEEPDEEP bit + WFI, Return from ISR or WFE	WKUP pin rising edge, RTC alarm (Alarm A or Alarm B), RTC Wakeup event, RTC tamper event, RTC timestamp event, external reset in NRST pin, IWDG reset			OFF

Figure 10: Summary of low power modes in a STM32L073 (Reference Manual)

2.2.1 Entering and exiting low power mode

The Figure 9 and Figure 10 give the action to enter each low power mode (column Entry), and the actions which wakes up the processor (column Wakeup). We can notice that the deeper is the low power mode, the fewer are wakeup possibilities.

For entering a low power mode, we have to use one of these instructions or features combines with some bits configuration:

- the wfi (Wait For Interrupt) instruction
- the wfe (Wait For Event) instruction
- the "Sleep on Exit" feature

2.2.2 Normal Sleep Mode

The Normal Sleep mode stops the processor clock, but all peripherals keep on running. On the STM32 clock tree (Figure 11), that means:

- FCLK (Cortex Clock) stops
- All the other activated clocks run

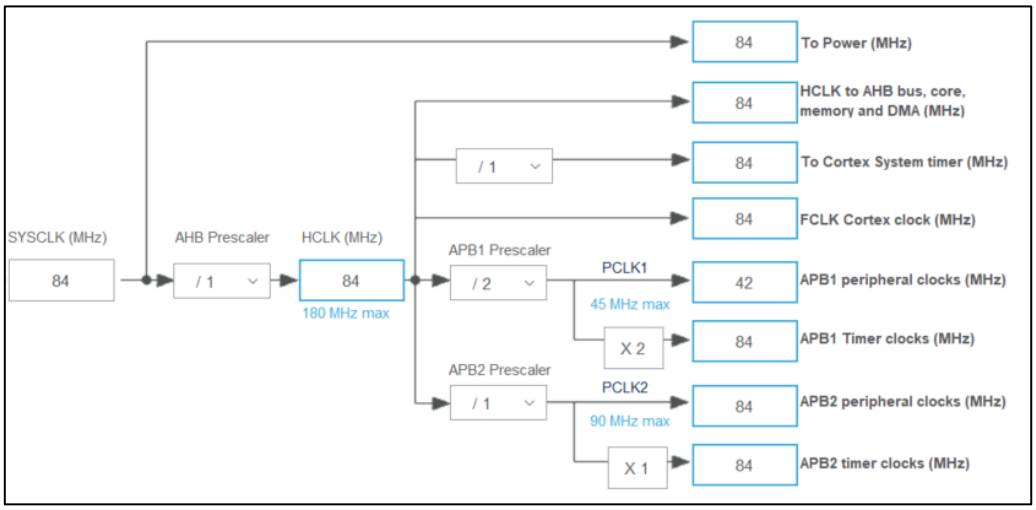


Figure 11: Clock tree in STM32CubeMX

➡ Create the following application on the STM32F446, which goes in low power mode (Normal Low Power - sleep) after each loop, using the wfi instruction.

Function	Code

main( )	`while (1) {`
	`  HAL_Delay(1000);`
	`  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);`
	`  HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);`
	`}`

What do we see?

At the first sight, we don't see much difference and we don't really see that the microcontroller goes into low power mode. That is because according to the Figure 9, the wakeup condition is "Any interrupt". On the cortex M microcontroller, the SysTick Timer lauches a permanent interrupt every millisecond (by default), so our application will jump out from low power mode only one millisecond after entering it !

You can only see this behaviour if you choose the right sampling frequency in STM32CubeMonitor-Power and if you zoom in the right area. It's also a good idea to reduce the value of the HAL_Delay in the application (10 instead of 1000 for example).

Function	Code
main( )	`while (1) { HAL_Delay(10); HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); }`

You should approximately find the following values (Table 3).

Test Conditions	STM32F446	STM32L073
Default Mode (CubeMX) Led ON / Running	20 000 µA	3 175 µA
Default Mode (CubeMX) Led ON / Low Power	11 900 µA	2810 µA
Default Mode (CubeMX) Led OFF / Running	17 000 µA	600 µA
Default Mode (CubeMX) Led OFF / Low Power	9 000 µA	250 µA

Table 3: Current consumption of STM32 F4/L0 in different states

Results with the STM32F446:

First, it is interesting to notice that the GPIO are still working while the processor is in Normal Sleep Mode. Indeed, the first state of the diagram below (LED ON / Low Power) shows that the processor is in Low Power, but is still powering the LED through its GPIO.

Figure 12: Normal Sleep mode in a STM32F446 microcontroller

Result with the STM32L073:

We expect the same behaviour with the STM32L073, but obviously with different consumption values.

2.2.3 Deep Sleep Mode

The Deep Sleep Mode stops the system Clock (SYSCLK) and switches off the PLL and flash memory. ST extends the Deep Sleep Mode in two modes which depend on the PDDS bit (**P**ower **D**own **D**eep **S**leep bit):

- Stop mode
- Standby mode

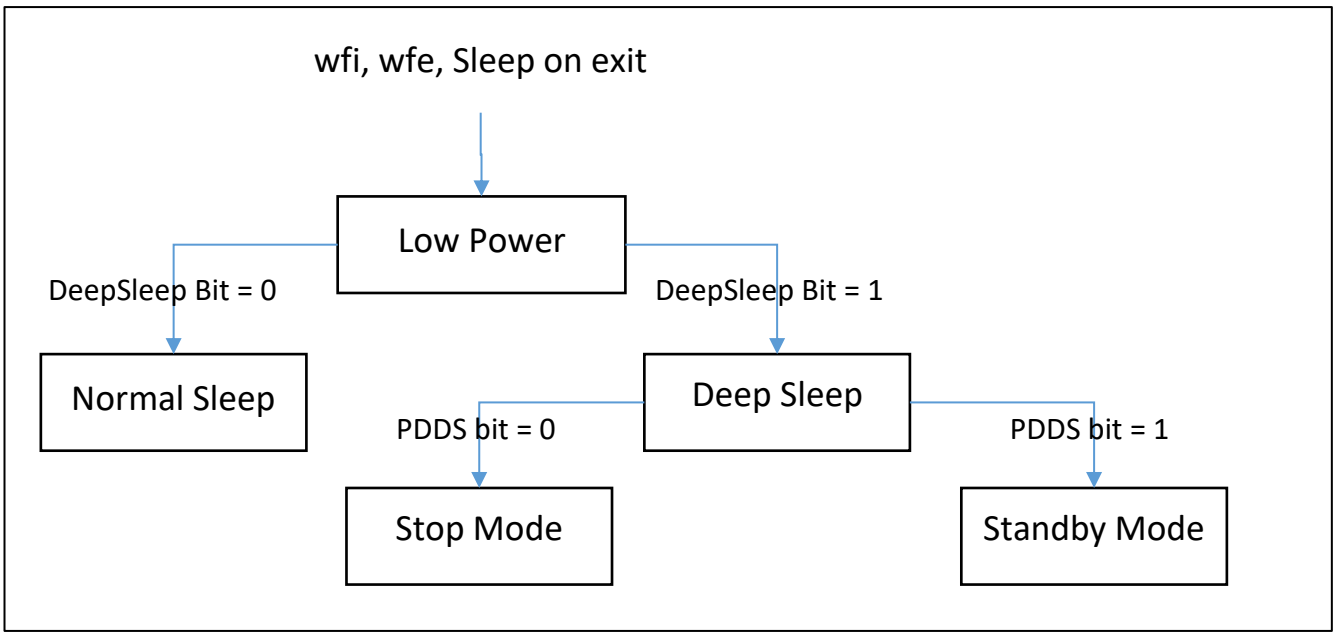


Figure 13: The Stop and Standby modes

In STM32Lx series, there are even more modes as you saw in Figure 10. Obviously, the deeper is the low power mode, the less power is consumed. However, there are two main drawback:

- There are less possibilities to wake up the microcontroller.
- The microcontroller takes longer to wake up.

We are not going to measure the power consumption in deep sleep now. Because whatever mode we are using, we have a lot to improve in our application before working on the deep sleep mode. Indeed, a low power application should never be waiting (HAL_delay) in running mode. In the next chapter, we will reorganize our code and present a new application based on interrupts and we will remove the LED, which consumes too much and prevents us to read very low current in deep sleep mode.

3 Reducing the power consumption

3.1 Initial application

Our initial application is not using any low power features. We will implement them in the next chapter and compare the current consumption after each improvement. Our simple application will write the text "Test of Low Power Mode on STM32" on the USART2 every 10 ms (interrupts are generated by TIM6 / APB1). The USART2 (APB1) baud rate is at 115200 bauds.

Function	Code

main( )	`HAL_TIM_Base_Start_IT(&htim6);`
	`While(1){`
	`// Nothing to do`
	`}`

TIM6 Interrupt routine	`uint8_t  textApp[]="Test of Low Power Mode on STM32\r\n";`
	
	`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){`
	`HAL_UART_Transmit(&huart2,textApp,sizeof(textApp),1000);`
	`}`

3.1.1 Timer configuration on the STM32F446

As we can see in the block diagram of the STM32F446 (Figure 67 in appendices), the APB1 Timer Clock runs the Timer 6. Its value is 84 MHz in the default configuration.

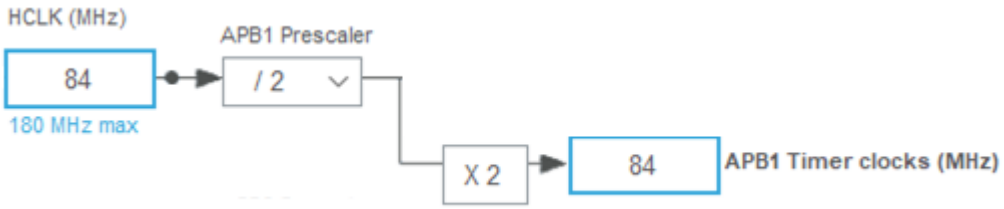


Figure 14: Default value of HCLK and APB1 Timer clock in STM32F446

The interrupt time is calculated by this formula:

$$Interrupt\ time = \frac{Prescaler + 1}{(Counter\ Period + 1)\ x\ f_{HCLK}} = 10\ ms$$

To have the 10ms interrupt with the STM32F446, we need to configure the TIM6 with the following values:

- Prescaler = 41999
- Counter Period = 19

3.1.2 Timer configuration on the STM32L073

As we can see in the block diagram (Figure 68 in appendices), the APB1 Timer Clock runs the Timer 6. Its value is 2.097 MHz in the default configuration.

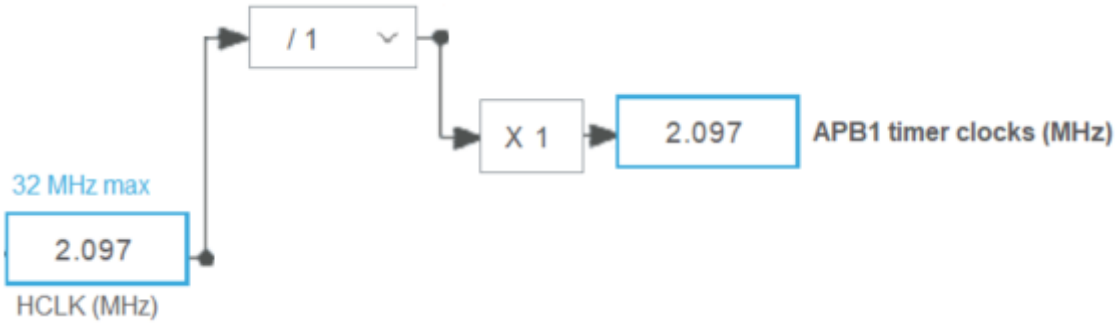


Figure 15: Default value of HCLK and APB1 Timer clock in STM32L073

The interrupt time is calculated by this formula:

$$Interrupt\ time = \frac{Prescaler + 1}{(Counter\ Period + 1)\ x\ f_{HCLK}} = 10\ ms$$

To have the 10ms interrupt with the STM32L073, we need to configure the TIM6 with the following values:

- Prescaler = 2096
- Counter Period = 9

3.1.3 Measurements

In this application, the microcontroller never goes in Low Power Mode. You can find the following average consumptions for both microcontrollers.

Test Conditions	Microcontroller	Sleep Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	None	16,95 mA
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI)	STM32L073	None	520 µA

Table 4: Average current consumption without Sleep mode

What do we see?

Every 10ms, the message "Test of Low Power Mode on STM32" is printed out on the serial link. During the transmission, the power consumption increases.

Figure 16: Power consumption on STM32F446 with initial values

3.2 Using the "Sleep On Exit" Feature

3.2.1 What is the Sleep On Exit Feature

The "Sleep On Exit" is one of the three ways to enter the Sleep mode. ARM Cortex processors have a feature which allow the processor to enter Sleep Mode as soon as the MCU exits an ISR (**I**nterrupt **S**ub-**R**outine).

When and how to use it?

It is useful to use it only when the processor runs the whole application in an interrupt routine. As soon as the Sleep On Exit is enabled, any code outside the ISR will be ignored. There is no need of any instructions to use it. The processor enters itself in Sleep mode after the interrupt routine. However, we have to configure this feature at the start of the application by setting the SLEEPONEXIT bit (see ARM Generic User Guide) in the **System Control Register** presented in the Figure 8. We can use the HAL function **HAL_PWR_EnableSleepOnExit();** which simply sets the SLEEPONEXIT bit.

Function	Code

main( )	`HAL_TIM_Base_Start_IT(&htim6);`
	`HAL_PWR_EnableSleepOnExit();`
	
	`While(1){`
	`// Nothing to do`
	`}`

TIM6 Interrupt routine	`uint8_t  textApp[]="Test of Low Power Mode on STM32\r\n";`
	
	`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){`
	`HAL_UART_Transmit(&huart2,textApp,sizeof(textApp),1000);`
	`}`

It is important to configure the Sleep On Exit feature at the end of the initialization, otherwise an interrupt can occur, and we will never come back to the main function.

3.2.2 Initial application with SLEEPONEXIT

On our application, we are going to use the SLEEPONEXIT feature to enter a **Normal Sleep Mode** when returning from the interrupt.

- Add the HAL_PWR_EnableSleepOnExit() in you main, just before the while loop.

You can find the following average consumptions for both microcontroller.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	None	16,95 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	Sleep On Exit Normal Sleep Mode	11,49 mA
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI)	STM32L073	None	520 µA
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI)	STM32L073	Sleep On Exit Normal Sleep Mode	360 µA

Table 5: Average current consumption with Sleep mode (STM32CubeMX default configuration)

What do we see?

You should clearly see the execution of the UART transmission every 10 ms. Between each transmission, the microcontroller is in Low Power mode. On STM32F446, the average power consumption drops from 16,95 mA to 11,49 mA which is a huge improvement, without compromising the application.

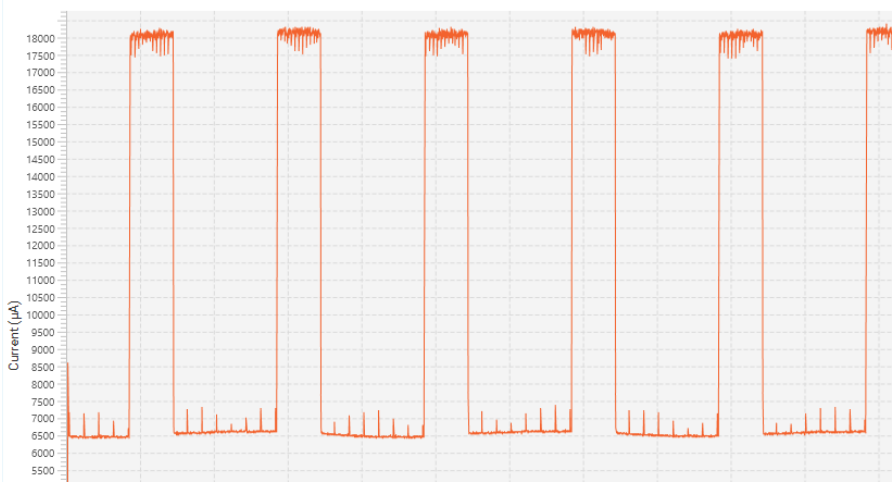


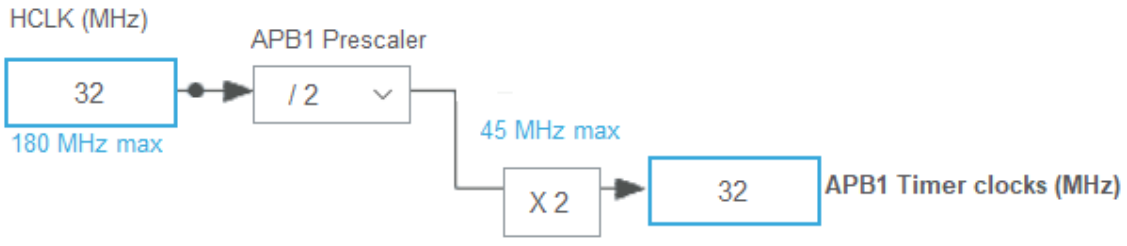
Figure 17: Current consumption in running and Low Power mode (STM32F446)

3.3 Effect of the CPU frequency

We keep the same applicatio, but we want to know how the clock frequency of the CPU affect the power consumption. We are going to change the HCLK clock at the following values:

- STM32F446: 2 MHz, 32 MHz, 84 MHz (Initial value), 180 MHz (max value).
- STM32L073: 2,097 MHz (initial value), 32 MHz (max value)

For comparison purpose, we will change only the HCLK Clock. You will therefore have to recalculate the TIM6 interrupt overflow value if the "APB1 Timer Clock" changes. For example, with the STM32F446, the initial HCLK value is 84 MHz and the APB1 Timer Clock is also 84 MHz (See Figure 14). When we change the HCLK clock value to 32 MHz, the APB1 Timer clock changes to 32 MHz.



$$Interrupt\ time = \frac{Prescaler + 1}{(Counter\ Period + 1)\ x\ f_{HCLK}} = 10\ ms$$

To have the 10ms interrupt with the STM32F446, we need to configure the TIM6 with the following values:

- Prescaler = 31999
- Counter Period = 9

You can find the following average consumptions for both microcontroller with and without the "Sleep On Exit" low power mode.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) Change HCLK to **180 MHz** (HSI)	STM32F446	None	38,22 mA
Default Mode (CubeMX) Change HCLK to **180 MHz** (HSI)	STM32F446	Sleep On Exit Normal Sleep Mode	23,71 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	None	16,95 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	Sleep On Exit Normal Sleep Mode	11,49 mA
Default Mode (CubeMX) Change HCLK to **32 MHz** (HSI)	STM32F446	None	8,46 mA
Default Mode (CubeMX) Change HCLK to **32 MHz** (HSI)	STM32F446	Sleep On Exit Normal Sleep Mode	6,48 mA
Default Mode (CubeMX) Change HCLK to **2 MHz** (HSI)	STM32F446	None	3,84 mA
Default Mode (CubeMX) Change HCLK to **2 MHz** (HSI)	STM32F446	Sleep On Exit Normal Sleep Mode	3,62 mA

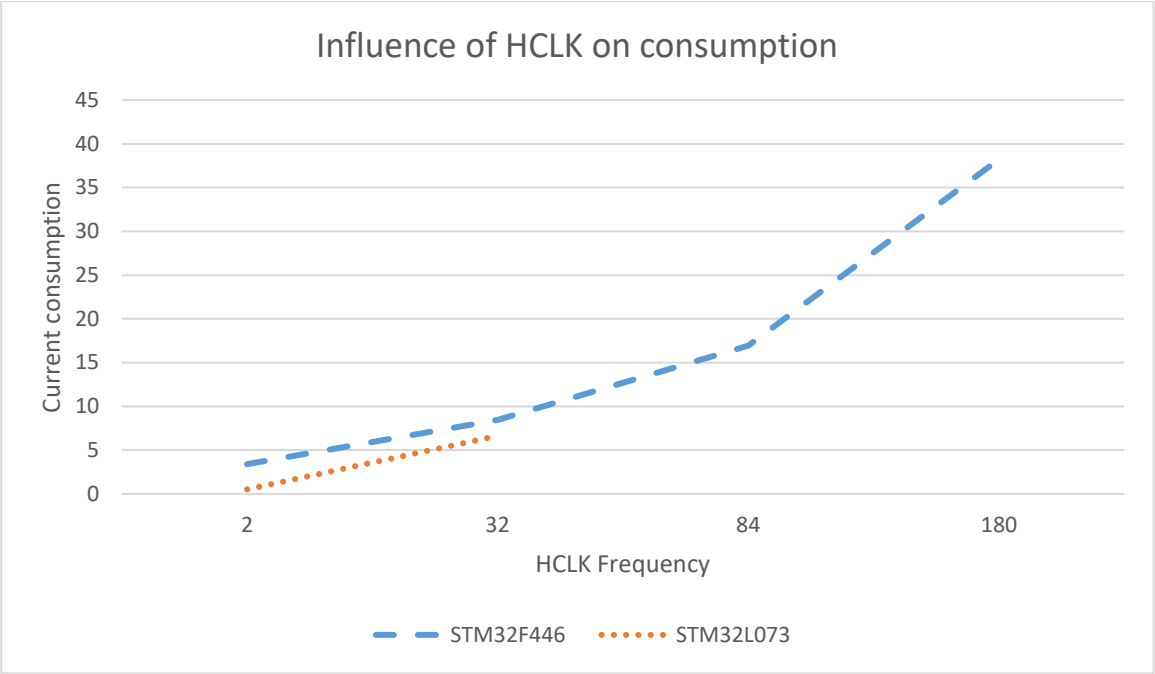
Table 6: Current consumption for different HCLK frequencies and sleep mode (STM32F446)

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) Change HCLK to **32 MHz** (HSE)	STM32L073	None	6,64 mA
Default Mode (CubeMX) Change HCLK to **32 MHz** (HSE)	STM32L073	Sleep On Exit Normal Sleep Mode	4,80 mA
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI)	STM32L073	None	520 µA
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI)	STM32L073	Sleep On Exit Normal Sleep Mode	360 µA

Table 7: Current consumption for different HCLK frequencies and sleep mode (STM32L073)

Note: We had to change the clock source from MSI to HSE to reach 32MHz on the STM32L073.

With all this values, we can plot the variation of the current consumption depending on the HCLK clock. The plot has been made when the Sleep On Exit features is disable.



What we should keep in mind?

We should always consider reducing the microcontroller clock to its minimum to improve the power consumption.

3.4 Effect of the temperature

The temperature has a real effect on the power consumption. It is difficult to measure it without a specific oven but the datasheet gives us interesting values.

Symbol	Parameter	Conditions	f_{HCLK} (MHz)	Typ	Max(2)			Unit
					T_A = 25 °C	T_A = 85 °C	T_A = 105 °C	
I_{DD}	Supply current in RUN mode	External clock, PLL ON, all peripherals enabled(3)(4)	180	86	93.0	115.0	125.0	mA
			168(5)	79	85.1	111.2	117.7	
			150	73	79.6	104.8	111.2	
			144(5)	68	73.5	97.3	103.3	
			120	54	59.3	79.7	84.7	
			90	42	47.23	65.50	70.10	
			60	29	33.7	49.5	53.4	
			30	16	20.8	34.0	37.4	
			25	13	18.4	31.2	34.5	
		HSI, PLL OFF, all peripherals enabled(3)(4)	16	8	13.8	25.0	28.3	
			8	5	10.8	21.1	24.2	
			4	3.0	9.1	18.9	22.0	
			2	2.1	8.1	17.8	20.9	
		External clock, PLL ON, all Peripherals disabled(3)	180	46	55.0	75.0	86.0	
			168	43	49.6	67.5	72.6	
			150	41	48.2	65.8	70.8	
			144(5)	38	43.6	61.9	66.8	
			120	32	37.3	53.7	58.0	
			90	26	30.7	46.0	50.0	
			60	18	22.8	36.4	40.1	
			30	10	14.9	27.1	30.2	
			25	9	13.55	25.40	28.54	
		HSI, PLL OFF, all peripherals disabled(3)	16	5	11.1	21.8	25.0	
			8	3	9.5	19.4	22.5	
			4	2.4	8.34	18.10	21.17	
			2	1.8	7.77	17.39	20.50	

Figure 18: Current consumption depending on the temperature (Datasheet STM32F446)

However, we can set up a rough test by using a heater or a basic hairdryer on the Nucleo board and notice that the power consumption increases at the same time as the temperature. The result is presented on the Figure 19. For this test, you can use any application. We simply used the STM32F446 with an empty while(1) loop, and without any peripheral or interrupt enabled.

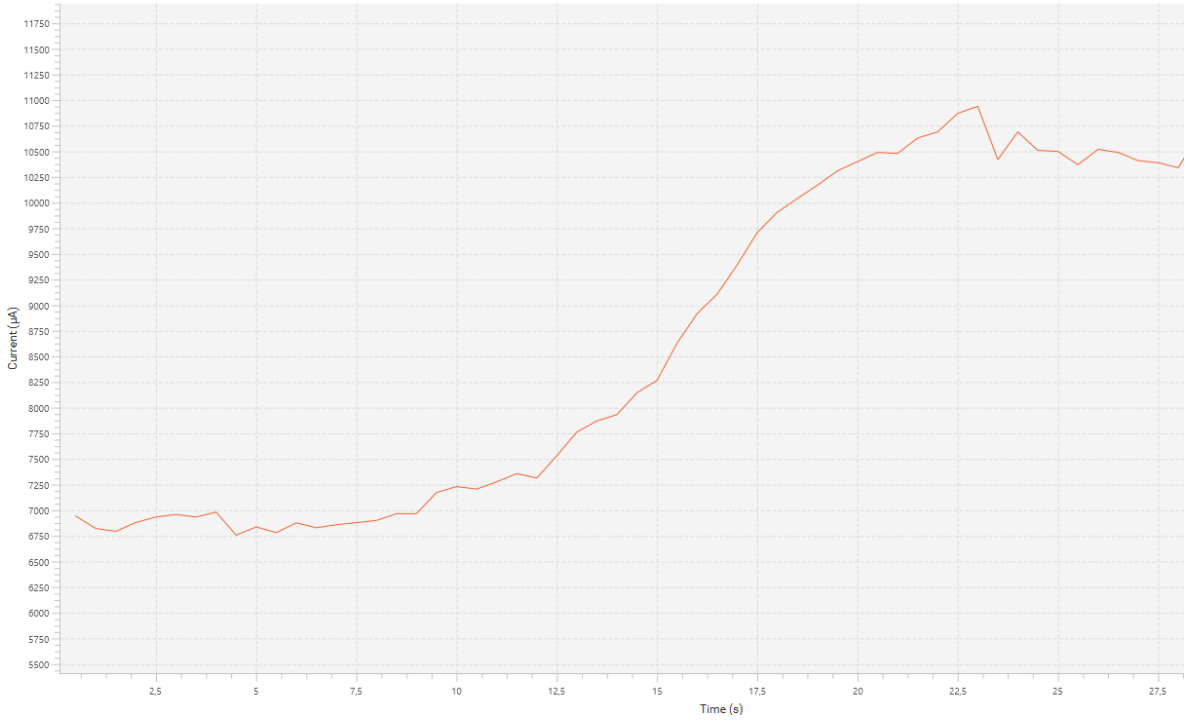


Figure 19: Raise of the current consumption when the temperature increases

What do we see?

We can clearly notice a raise of consumption when we provide heat on the microcontroller five seconds after the start.

3.5 Effect of the Clock Source

3.5.1 Clock sources on a STM32F446

Two different high-speed clocks can source the SYSCLK clock:

- HSI oscillator clock (**H**igh **S**peed **I**nternal)
- HSE oscillator clock (**H**igh **S**peed **E**xternal)

The RTC or watchdog can use other low speed clock:

- LSI oscillator clock (**L**ow **S**peed **I**nternal)
- LSE oscillator clock (**L**ow **S**peed **E**xternal)

3.5.2 Clock sources on a STM32L073

STM32L073 has exactly the same clock source than the STM32F446 plus some others:

Three different high-speed clocks can source the SYSCLK clock:

- MSI oscillator clock (**M**ulti**S**peed **I**nternal)
- HSI oscillator clock (**H**igh **S**peed **I**nternal)
- HSE oscillator clock (**H**igh **S**peed **E**xternal)

The RTC or watchdog can use other low speed clock:

- LSI oscillator clock (**L**ow **S**peed **I**nternal)
- LSE oscillator clock (**L**ow **S**peed **E**xternal)

The specific USB peripheral can have its own internal clock:

- RC48

3.5.3 HSI clock

On the STM32F446 and STM32L073 microcontrollers, HSI is a 16MHz internal RC oscillator. The system clock can use it directly, or through PLL.

- Advantage: low cost (no need of crystal) and faster startup time than the HSE.
- Drawback: less accurate.

The STM32F446 datasheet provide the HSI accuracy (Figure 20).

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
f_{HSI}	Frequency	-	-	16	-	MHz
ACC_{HSI}	Accuracy of the HSI oscillator	User-trimmed with the RCC_CR register(2)	-	-	1	%
		T_A = - 40 to 105 °C(3)	- 8	-	4.5	%
		T_A = - 10 to 85 °C(3)	- 4	-	4	%
		T_A = 25 °C(4)	- 1	-	1	%
$t_{su(HSI)}$(2)	HSI oscillator startup time	-	-	2.2	4	µs
$I_{DD(HSI)}$(2)	HSI oscillator power consumption	-	-	60	80	µA

Figure 20: HSI Oscillator characteristics (Datasheet STM32F446)

For example, we want to know how much time a clock would derive during one day if we consider an ambient temperature of 25 °C when using the HSI clock.

Answer: At 25°C, the accuracy is 1%. There is 24x3600=86400 seconds in on day. The clock will derive of 86400x0.01=864 seconds in on day, hence 14'24'' per day.

3.5.4 HSE clock

HSE is an external clock, which can be generated from:

- An external user clock (default case of the Nucleo board)
- An external crystal or ceramic resonator

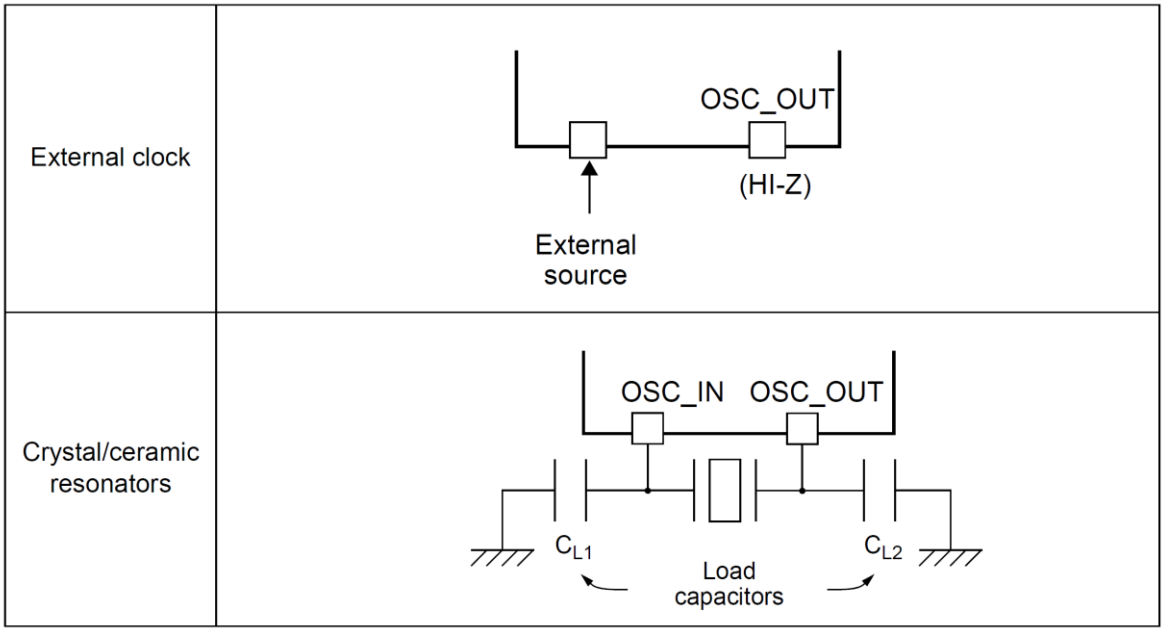


Figure 21: Clock sources (HSE) - Reference Manual STM32F466

- Advantage: As accurate as the external component is (crystal or ceramic resonator)
- Drawback: Expensive, higher startup time than HSI

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
f_{OSC_IN}	Oscillator frequency	-	4	-	26	MHz
R_F	Feedback resistor	-	-	200	-	kΩ
I_{DD}	HSE current consumption	V_{DD}=3.3 V, ESR= 30 Ω, C_L=5 pF@25 MHz	-	450	-	µA
		V_{DD}=3.3 V, ESR= 30 Ω, C_L=10 pF@25 MHz	-	530	-	
ACC_{HSE} (2)	HSE accuracy	-	-500	-	500	ppm
G_m_crit_max	Maximum critical crystal g_m	Startup	-	-	1	mA/V
$t_{SU(HSE}$ (3)	Startup time	V_{DD} is stabilized	-	2	-	ms

Figure 22: HSE oscillator characteristics – datasheet STM32F446

On our the Nucleo board, the 8 MHz oscillation on HSE is provided by the ST-Link µC. There is no crystal on the board but the footprint is available.

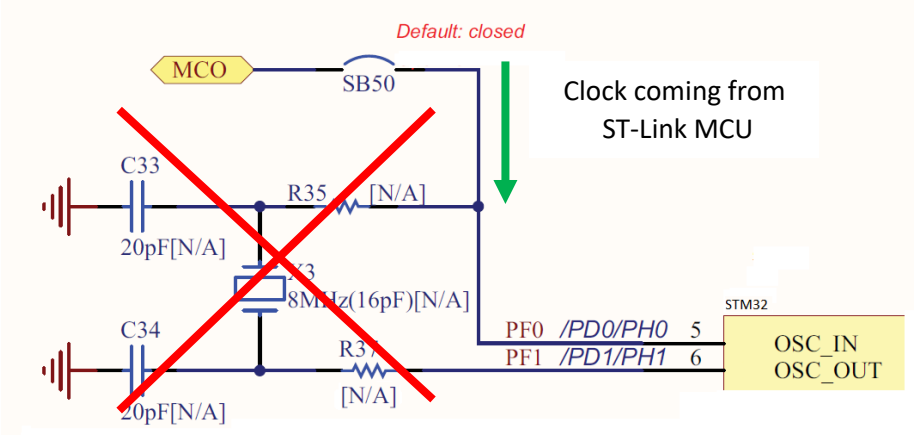


Figure 23: HSE configuration on the Nucleo board

With this Nucleo board, we are using the "External User clock" mode (as explained in Figure 21). This mode is selected in CubeMX in the configuration of the STM32 in **System Core > RCC > Mode > High Speed Clock (HSE) > Bypass Clock source.**

On our Nucleo board, the X3 oscillator can run:

- From 4 to 26 MHz on the STM32F446
- From 0 to 32 MHz on the STM32L073

Crystals can be a very expensive component in an electronic design. We have to know the accuracy we want before buying it. If we check on 4 different crystals the average value of the frequency tolerance, the price, and how much time a clock could derive during one day, we have to following results with the following components:

1. a 16 MHz ceramic resonator
2. a 16MHz crystal
3. a 16MHz crystal oscillator TCXO (**T**emperature **C**ompensate **X**tal **O**scillator)
4. a 16MHz crystal oscillator OCXO (**O**ven **C**ontroled **X**tal **O**scillator)

Clock Source	Approx unit price	Tolerance	Time shift (1day)
1. Ceramic resonator	0,25€	0.5%	7 min 12s
2. Crystal	0,4 € (bigger)	30 ppm	2,6 s
3. Crystal oscillator (TCXO)	2 €	2,5 ppm	216 ms
4. Crystal oscillator (OCXO)	100 €	20 ppb	1,73 ms

Table 8: Approximate price and Tolerance of different clock source

3.5.5 LSI

The LSI is an RC oscillator, which can still run even in Stop and Standby mode for the independent watchdog (auto-wake up). The clock frequency is around 32 kHz and the accuracy is very poor.

Table 42. LSI oscillator characteristics (1)

Symbol	Parameter	Min	Typ	Max	Unit
f_{LSI} (2)	Frequency	17	32	47	kHz
$t_{su(LSI)}$ (3)	LSI oscillator startup time	-	15	40	µs
$I_{DD(LSI)}$ (3)	LSI oscillator power consumption	-	0.4	0.6	µA

Table 9: LSI oscillator characteristics

3.5.6 LSE

The LSE clock can be a low-speed crystal or a ceramic resonator. Its common value is 32,768 kHz.

On the Nucleo board STM32L073, the LSE crystal reference is "ABS25-32.768KHZ-6-T". The Table 10 summarizes its features.

Component	Approx unit price	Tolerance	Dimension
ABS25-32.768KHZ-6-T	0,1€	20 ppm	8 mm/ 3mm

Table 10: Characteristics of the ABS25-32.768KHZ-6-T Crystal

3.5.7 MSI (Multi Speed Internal) Clock

The MSI clock is an internal RC oscillator. Its frequency range is tuned by software. Seven frequencies are available from 65 536 kHz to 4.194 MHz.

Why the MSI?

Indeed, it has the same purpose as the HSI, which is also a RC oscillator. If we look at the datasheet we can learn that the MSI can be used at lower power mode than the HSI (**O**: **O**ptional, **Y**: **Y**es, **--**: Not Available).

IPs	Run/Active	Sleep	Low-power run	Low-power sleep	Stop		Standby	
						Wakeup capability		Wakeup capability
High Speed Internal (HSI)	O	O	--	--	(3)		--	
High Speed External (HSE)	O	O	O	O	--		--	
Low Speed Internal (LSI)	O	O	O	O	O		O	
Low Speed External (LSE)	O	O	O	O	O		O	
Multi-Speed Internal (MSI)	O	O	Y	Y	--		--	

Figure 24: Clock availability depending on the working modes

But the MSI oscillator is even worse than HSI about accuracy, so you need to consider this to select the right clock source between HSI and MSI.

3.5.8 HSI48 (High Speed Internal 48 MHz) Clock

HSI48 is an internal RC oscillator used for USB purposes.

3.5.9 Power consumption

We can make some measurement with difference clock source. For that purpose, we will use the STM32L073. We first remind the previous values measured for our application.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI) **(1)**	STM32L073	None	520 µA
Default Mode (CubeMX) with HCLK to **2.097 MHz** (MSI) **(1)**	STM32L073	Sleep On Exit Normal Sleep Mode	360 µA

Table 11: Current consumption depending on the clock used

Then we change the clock sources and use the path (2), (3), (4) and (5).

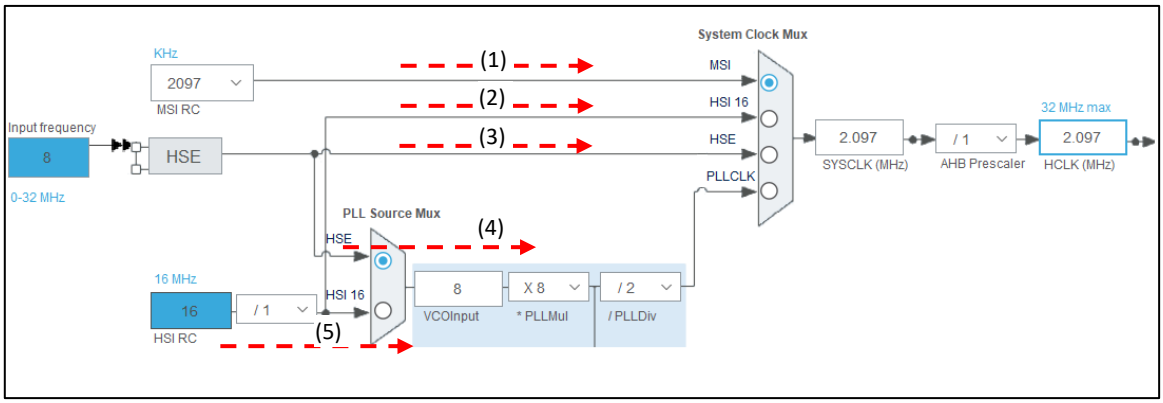


Figure 25: Clock sources for the STM32L073

The values are stored in the table below. We use a 2MHz clock because it is not possible to generate 2.097 MHz. Don't forget to change the TIM6 Interrupt time with the right values (Prescaler = 1999 and Counter Period = 9).

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) With HCLK to **2 MHz** (HSI 16 from HSI RC) **(2)**	STM32L073	Sleep On Exit Normal Sleep Mode	860 µA
Default Mode (CubeMX) With HCLK to **2 MHz** (HSE from HSE) **(3)**	STM32L073	Sleep On Exit Normal Sleep Mode	500 µA
Default Mode (CubeMX) With HCLK to **2 MHz** (PLLCLK from HSE) **(4)**	STM32L073	Sleep On Exit Normal Sleep Mode	700 µA
Default Mode (CubeMX) With HCLK to **2 MHz** (PLLCLK from HSI RC) **(5)**	STM32L073	Sleep On Exit Normal Sleep Mode	1200 µA

What we shall keep in mind?

The clock source has an effect on power consumption. We can summarize it in a table.

	MSI	HSI	HSE
Consumption	Excellent	High power	Average
Accuracy	Very poor	Poor	Good
Cost	No additional cost	No additional cost	Need an oscillator

We can also notice that using the PLL increase the power consumption.

3.6 Effect of the USART2 Baudrate

We can see on the STM23CubeMonitor-Power the moment when the CPU is using the UART and when it is entering the low power mode. Transferring the data faster with the UART will forward the moment the CPU is entering the low power mode. Therefore, it will save more power.

On the default configuration (HCLK = 84 MHz / HSI) on the STM32F446 MCU, we are going to increase the Baudrate to 230400 bauds and compare with the previous power consumption. Here are the previous measurement:

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **115200**	STM32F446	None	16,95 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **115200**	STM32F446	Sleep On Exit Normal Sleep Mode	11,49 mA

Table 12: Previous measurement with default configuration.

New measurement values are stored in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400**	STM32F446	Sleep On Exit Normal Sleep Mode	9,32 mA

Current consumption depending on the USART2 Baud Rate

We could even go faster and increase the Baudrate but for the next step, we will keep the Baudrate to 230400.

What do we see?

We can notice that the MCU spends less time sending data through the USART, so the average current is reduced.

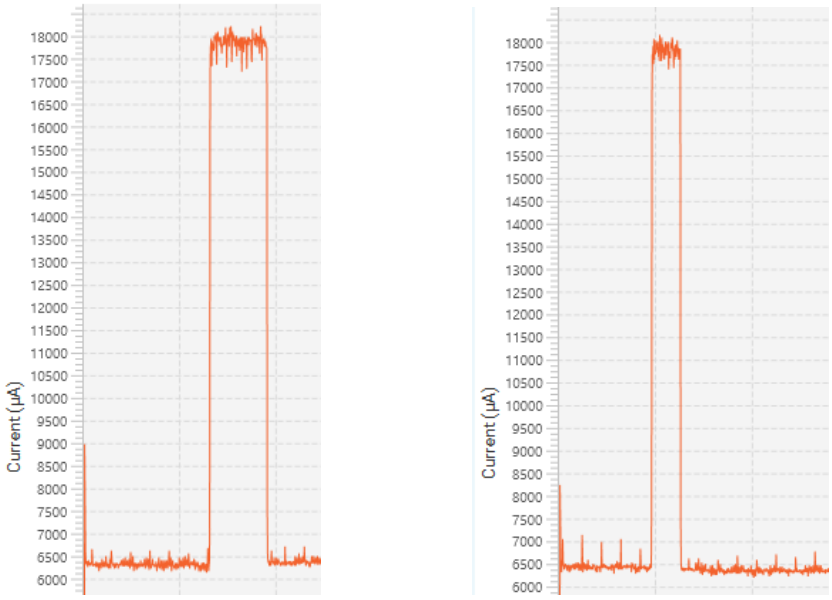


Figure 26: Transmission time at 115200 (left) and 230400 (right) bauds

3.7 Effect of the APB1 peripheral frequency

So far, we have seen that we can reduce the power consumption if we lower the clock frequency (HCLK) for the CPU. There is the same possibility for each enabled peripherals. We are using the USART2 at 230400 bauds and the USART2 work with the APB1 bus. By default, the "APB1 Peripheral Clock" is at 42 MHz.

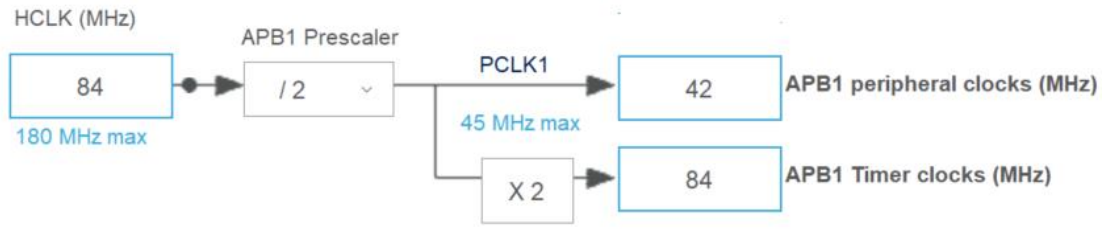


Figure 27: The actual APB1 peripheral clock frequency (STM32CubeMX)

We can reduce the "APB1 Peripheral Clock" at 5.25 MHz. The maximum Baudrate is now limited but we can still reach 230400 bauds.

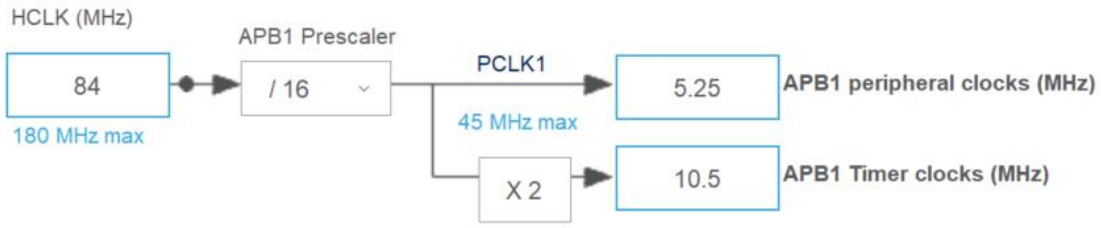


Figure 28: The target APB1 peripheral frequency (CubeMX)

The TIM6 clock has also change, so we also have to change the TIM6 Prescaler (10499) and the TIM6 Counter Period (9).However, the overall power consumption is now reduced as we can see in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **42 MHz**	STM32F446	Sleep On Exit Normal Sleep Mode	9,32 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz**	STM32F446	Sleep On Exit Normal Sleep Mode	7,79 mA

Table 13: Power consumption depending on the peripheral clock frequency (APB1)

3.8 Effect of the USART2 mode

The USART2 peripheral is configured with the RX and TX capabilities.

Figure 29: CubeMX configuration of the USART2

However, in our application, we are not using the reception. Therefore, we can disable it in order to remove the clock and the power to the reception hardware part. It is not straightforward to see the improvement in power consumption, so we can say that it is not a major issue.

New measurement values are stored in the Table 14.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Receive** and **Transmit**	STM32F446	Sleep On Exit Normal Sleep Mode	7,79 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only	STM32F446	Sleep On Exit Normal Sleep Mode	7.75 mA

Table 14: Current consumption depending on the USART2 configuration

3.9 Effect of USART2 Clock gating

It is up to the programmer to enable the peripherals when the application needs it. When, we don't need it any more, it worth to disable it by removing its clock. This action is called "clock gating".

Which clock can we stop in our application?

- About The TIM6 clock: We cannot stop this clock because this peripheral is always running (either during the interruption or during the sleep mode). No clock gating is applicable on this peripheral.

- About The USART2: We use the USART2 only during the ISR, so we can try to disable this clock before entering the Sleep mode, and enable it back again when waking up from the sleep mode, just before sending the data.

We could use the HAL function __HAL_RCC_USART2_CLK_ENABLE() and __HAL_RCC_USART2_CLK_DISABLE() to enable and disable the peripheral clock. The nice thing with STM32 is that it automatically runs this behaviour if we use the "**RCC APB1 peripheral clock enable in low power mode**" register. This allows the processor to automatically stop the USART2 clock when entering the low power mode and start it again when going back in running mode.

6.3.18 RCC APB1 peripheral clock enable in low power mode register (RCC_APB1LPENR)

Address offset: 0x60

Reset value: 0x3FFF C9FF

Access: no wait state, word, half-word and byte access.

Bit 17 **USART2LPEN:** USART2 clock enable during Sleep mode
This bit is set and cleared by software.
0: USART2 clock disabled during Sleep mode
1: USART2 clock enabled during Sleep mode

Figure 30: The RCC APB1 peripheral clock enable in low power mode register

This bit is configured by the Macro: **__HAL_RCC_USART2_CLK_SLEEP_DISABLE()**

Function	Code

main( )	`__HAL_RCC_USART2_CLK_SLEEP_DISABLE();`
	`HAL_TIM_Base_Start_IT(&htim6);`
	`HAL_PWR_EnableSleepOnExit();`
	`While(1){`
	`// Nothing to do`
	`}`

New values are stored in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only **No** Clock Gating	STM32F446	Sleep On Exit Normal Sleep Mode	7.79 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only Clock Gating on **USART2**	STM32F446	Sleep On Exit Normal Sleep Mode	7.77 mA

Table 15: Current consumption with or without clock gating on USART2

3.10 Effect of the GPIO configuration

When the GPIO are unused, we usually set the GPIO pin in digital input mode. But by using analog input instead of digital input, we can save more power because that disables the Smith Trigger and therefore reduce the overall power consumption.

- Keep the unused pin as analog: **CubeMX > Project Manager Tab > Code Generator > Hal_Settings > Set all free pin as analogs.**

New values are stored in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only Clock Gating on **USART2** Unsuded PINs as Digital INPUT	STM32F446	Sleep On Exit Normal Sleep Mode	7.77 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only Clock Gating on **USART2** Unused pin as **analog**	STM32F446	Sleep On Exit Normal Sleep Mode	6.85 mA

Table 16: Current consumption depending on the unused GPIO configuration

3.11 Effect of the SysTick interrupt

In our application, the SysTick Timer wakes up the MCU every 1 ms without doing any relevant actions. The Figure 31 shows the consumption peaks at each wake up. The behaviour is very short and the sampling time of our monitor is probably not fast enough to catch the entire pulse.

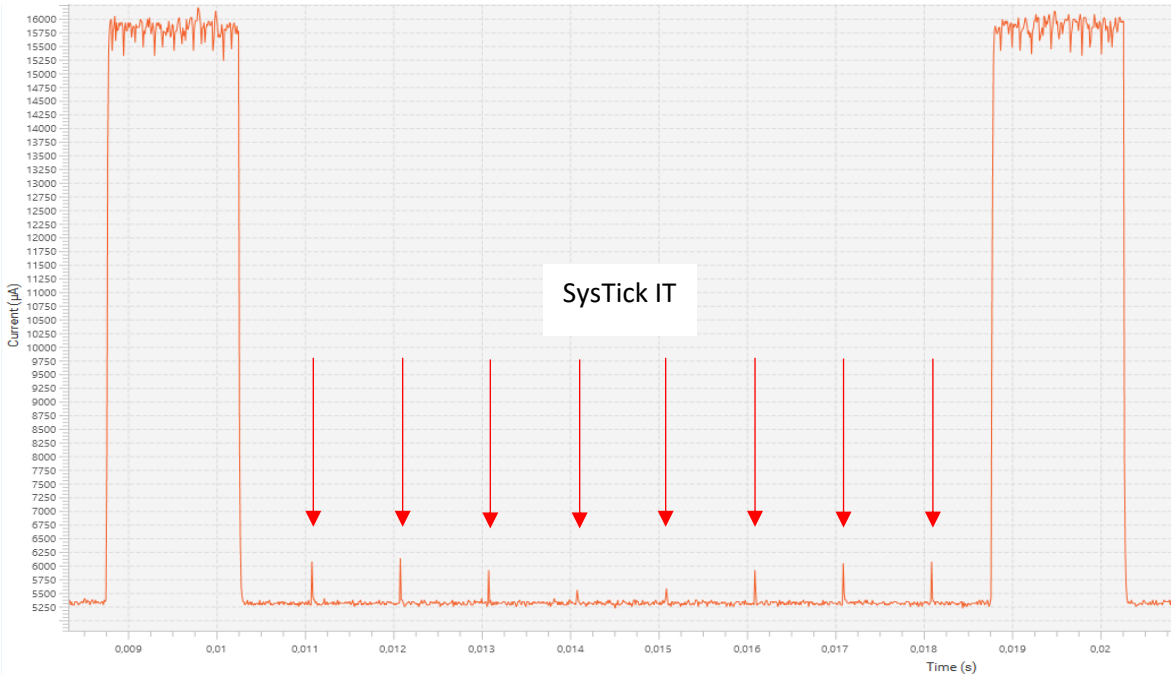


Figure 31: SysTick IT every 1 ms (STM32F446)

We could suspend the SysTick Timer before going into the Sleep Mode, and resume it when the application is back in running mode. For that purpose, we use the following HAL functions.

- **HAL_SuspendTick();** // At the end of the interrupt
- **HAL_ResumeTick();** // At the beginning of the interrupt

Function	Code

TIM6 Interrupt routine	`uint8_t  textApp[]="Test of Low Power Mode on STM32\r\n";`
	
	`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){`
	`  HAL_ResumeTick();`
	`  HAL_UART_Transmit(&huart2,textApp,sizeof(textApp),1000);`
	`  HAL_SuspendTick();`
	`}`

We can see that there is no Tick IT anymore during the sleep mode. New consumption values are stored in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only Clock Gating on **USART2** Unused pin as **analog** SysTick always **ON**	STM32F446	Sleep On Exit Normal Sleep Mode	6.85 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only Clock Gating on **USART2** Unused pin as **analog** SysTick **OFF** during Sleep Mode	STM32F446	Sleep On Exit Normal Sleep Mode	6.55 mA

Table 17: Current consumption depending on SysTick IT in Sleep Mode (STM32F446)

3.12 Effect of using USART Interrupt

In our application, each time we send a byte through the USART, we wait for the transfer to be completed in pooling mode. This is a waste of time because the MCU stays in Running mode whereas it should wait in Low power mode. We will use the USART interrupt to wake up the MCU as soon as a new data is ready to be transmitted. The STM32 will spend most of its time in Low Power mode.

However, we need to change our previous Low Power configuration because when the STM32 goes to sleep, we configured earlier the clock gating on the USART2. Which means that the USART was disabled in Sleep mode. That is not the case anymore. So, for the next experiment, we will remove the clock gating and suppress the function **__HAL_RCC_USART2_CLK_SLEEP_DISABLE()** in our code.

Function	Code

main( )	`HAL_TIM_Base_Start_IT(&htim6);`
	`HAL_PWR_EnableSleepOnExit();`
	
	`While(1){`
	`// Nothing to do`
	`}`

The USART2 interrupt needs to be enabled in CubeMX: **USART2 > NVIC Settings > USART2 global IT**.

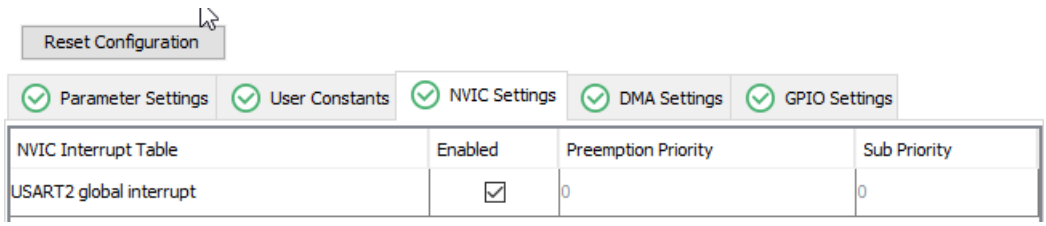


Figure 32: USART2 IT configuration

The ISR also needs to be updated to call the UART_Transmit_IT() instead of UART_Transmit().

Function	Code

TIM6 Interrupt routine	`uint8_t  textApp[]="Test of Low Power Mode on STM32\r\n";`
	
	`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){`
	`  HAL_ResumeTick();`
	`  HAL_UART_Transmit_IT(&huart2,textApp,sizeof(textApp));`
	`  HAL_SuspendTick();`
	`}`

New consumption values are stored in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only Clock Gating on **USART2** Unused pin as **analog** SysTick **OFF** during Sleep Mode USART Tx **without interrupt**	STM32F446	Sleep On Exit Normal Sleep Mode	6.55 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only ~~Clock Gating on **USART2**~~ Unused pin as **analog** SysTick **OFF** during Sleep Mode USART Tx **with interrupt**	STM32F446	Sleep On Exit Normal Sleep Mode	5.64 mA

Table 18: Power consumption with or without using IT on USART2 TX

What do we see?

During transfer, we can clearly see that the processor exits from Sleep Mode as many times as there are bytes to send. The running time is reduce, and so is the power consumption.

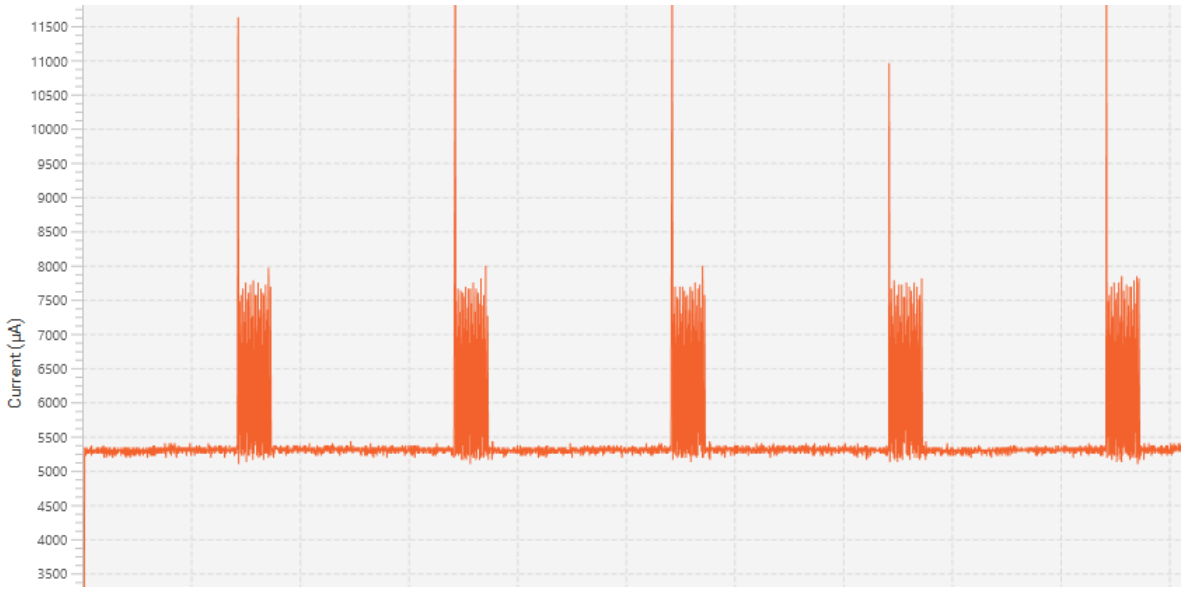


Figure 33: USART2 TX with interrupt (STM32F446)

3.13 Effect of the DMA for sending the data

When we use the USART2 in interrupt mode for sending data, the USART interrupts the CPU as many times as there are bytes to transmit. This job is typically what a DMA can perform without disturbing the CPU. In that experiment, everything will be done in Low Power mode while the DMA will deal with the byte transfer. The STM32 will be woken up only to launch the transfer, and to be noticed at the end of it.

The USART2 DMA needs to configured in CubeMX: **USART2 > DMA Settings**.

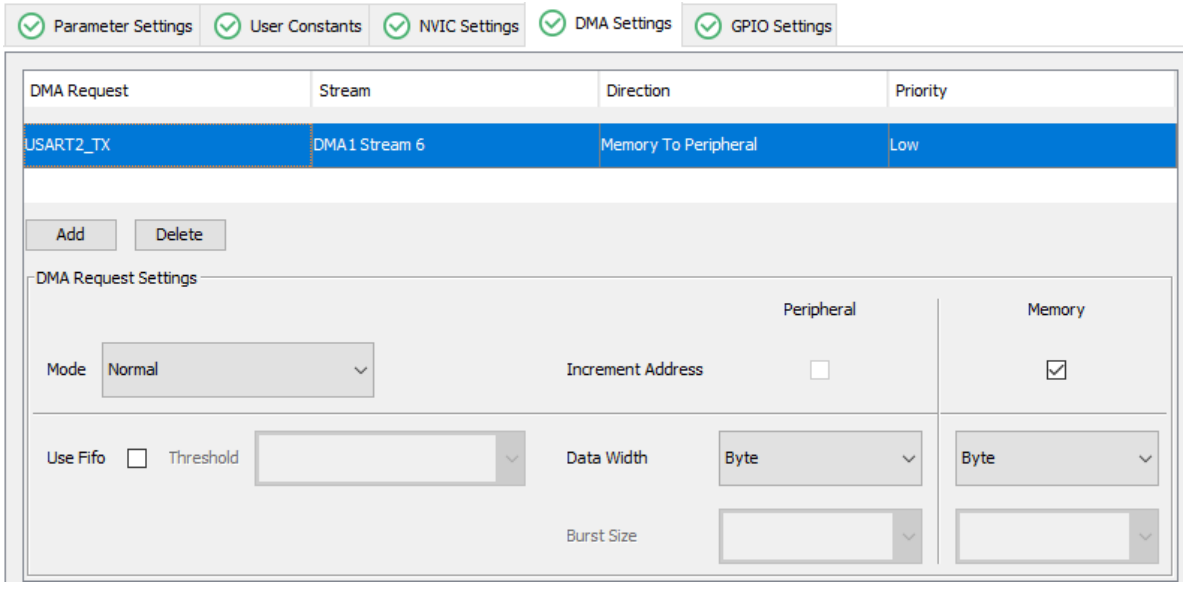


Figure 34: DMA configuration for USART2 TX

The USART2 DMA interrupt needs also to be enabled: **USART2 > NVIC Settings.**

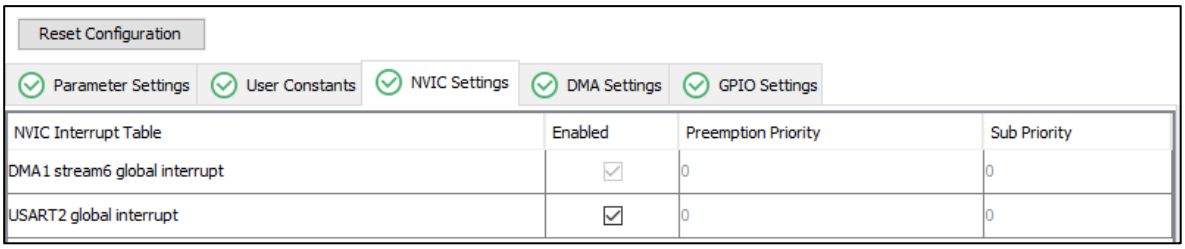


Figure 35: USART2 TX DMA interrupt configuration

The ISR also needs to be updated to call the UART_Transmit_DMA() instead of UART_Transmit_IT().

Function	Code

TIM6 Interrupt routine	`uint8_t  textApp[]="Test of Low Power Mode on STM32\r\n";`
	
	`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){`
	`  HAL_ResumeTick();`
	`  HAL_UART_Transmit_DMA(&huart2,textApp,sizeof(textApp));`
	`  HAL_SuspendTick();`
	`}`

New consumption values are stored in the table below.

Test Conditions	Microcontroller	Low Power Mode	Average consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only ~~Clock Gating on **USART2**~~ Unused pin as **analog** USART Tx **with interrupt**	STM32F446	Sleep On Exit Normal Sleep Mode	5.64 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **230400** APB1 Peripheral Clock = **5.25 MHz** **Transmit** only ~~Clock Gating on **USART2**~~ Unused pin as **analog** USART Tx **with DMA**	STM32F446	Sleep On Exit Normal Sleep Mode	5.80 mA

Table 19: Current consumption with IT or with DMA

What do we see?

The consumption is higher when using the DMA, so we don't really have a big advantage on power consumption. It can come from several explanations:

1. The power consumption with USART2 in interrupt is probably underestimated because the peaks of current are too short to be taken into account.
2. The DMA is a CPU peripheral, which needs to be powered and clocked. That induces a raise of power consumption.

We can see in the Figure 36 the step of current consumption of the DMA peripheral every 10 ms. We can also notice that the CPU has a short running period at the beginning, the half and the end of the transfer.

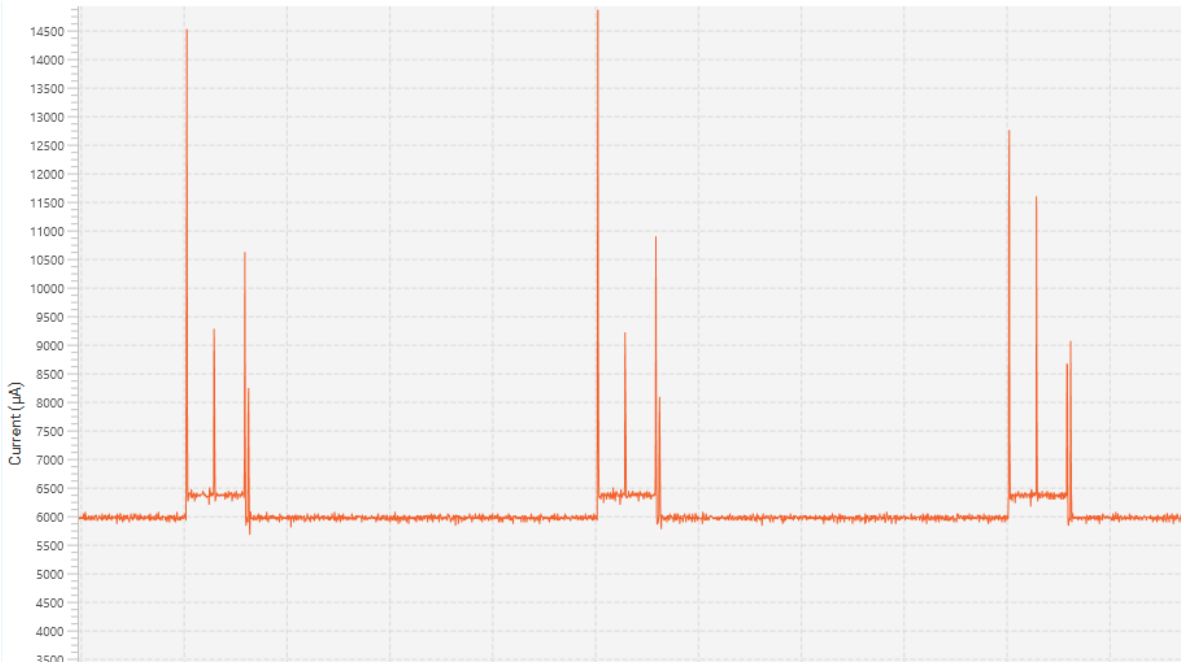


Figure 36: USART2 TX with DMA (STM32F446)

3.14 Effect of the Code optimization

So far, with optimization level "None (-O0)" we have to following results.

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH	0x08000000	0x08080000	512 KB	498,89 KB	13,11 KB	2.56%
RAM	0x20000000	0x20020000	128 KB	126,2 KB	1,8 KB	1.40%

Figure 37: Flash and RAM used with no optimization.

- Let's change the compiler optimization options on the STM32F446 and see if there are some improvements on calculation and therefore current consumption.

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH	0x08000000	0x08080000	512 KB	503,92 KB	8,08 KB	1.58%
RAM	0x20000000	0x20020000	128 KB	126,2 KB	1,8 KB	1.40%

Figure 38: Flash and RAM used with optimization for speed

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH	0x08000000	0x08080000	512 KB	504,49 KB	7,51 KB	1.47%
RAM	0x20000000	0x20020000	128 KB	126,2 KB	1,8 KB	1.40%

Figure 39: Flash and RAM used with optimization for size

I did not get any difference in power consumption for any optimization level. However, for some type of application, it probably worth trying it.

4 How to enter the Sleep Mode - WFI / WFE instruction

So far, we have seen the SLEEPONEXIT feature which enters/exits automatically the Sleep mode when an interrupt occurs. Now, we will see how we can launch the Sleep mode, and which events wake up the MCU.

4.1 WFI instruction: Wait For Interrupt

4.1.1 Entering the low power mode

For entering the low power mode, we just need to use the **wfi** instruction: it enters Sleep mode unconditionally, which means that any interrupt source will wake up the MCU. We can use the wfi instruction in handler mode (during an ISR) or in thread mode (in any other functions).

4.1.2 Our new Application

For this application, we will reset all previous configurations to its default state, as it is when you create a new project with STM32CubeIDE

The PC13 push button (see Figure 40) will interrupt the CPU and launch an USART2 data transmission (115200 Bauds) stating that the STM32 has been woken up. When the button is not pressed, the MCU executes all the functions in the while loop, than goes back in Sleep mode.

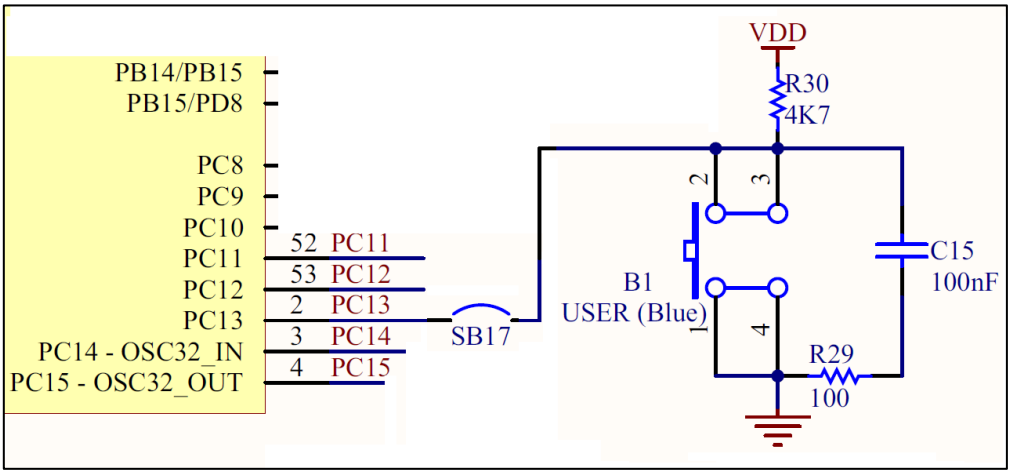


Figure 40: The User Button applies a falling edge when pressed

During our experiment, we will print some messages on the USART to understand how the application behaves. To make things easy, we will redirect the printf() function to the UART2.

- Write the following function between the "USER CODE BEGIN 0" and "USER CODE END 0" tags in your main.c files:

```
/* USER CODE BEGIN 0 */
int __io_putchar(int ch){
    HAL_UART_Transmit(&huart2, &ch, 1, 1000);
}
/* USER CODE END 0 */
```

- Add the #include<stdio.h> in the beginning of your main.c

We need to enable the falling edge interrupt on PC13 in CubeMX: **GPIO > NVIC** (see Figure 41).

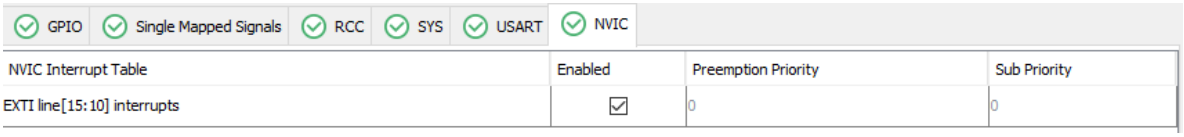


Figure 41: Falling edge interruption configuration for PC13 Push Button

The application will run the following code:

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	
	`while(1){`
	`printf("Running the while loop\r\n");`
	`printf("The processor goes to Normal sleep using wfi\r\n\r\n");`
	`HAL_SuspendTick();`
	`HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);`
	`}`

Push Button ISR	`void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){`
	`HAL_ResumeTick();`
	`printf("Wake Up by Push Button IT\r\n");`
	`}`

We must remember that the SysTick timer is still running and its interruption wakes up the MCU. To prevent that, we use the **HAL_SuspendTick**() function just before going in Sleep mode.

What do we see?

- The processor wakes up to execute its ISR than goes back to sleep.
- There is a step of current as long as the push button is pressed due to the R30 pull-up resistor.

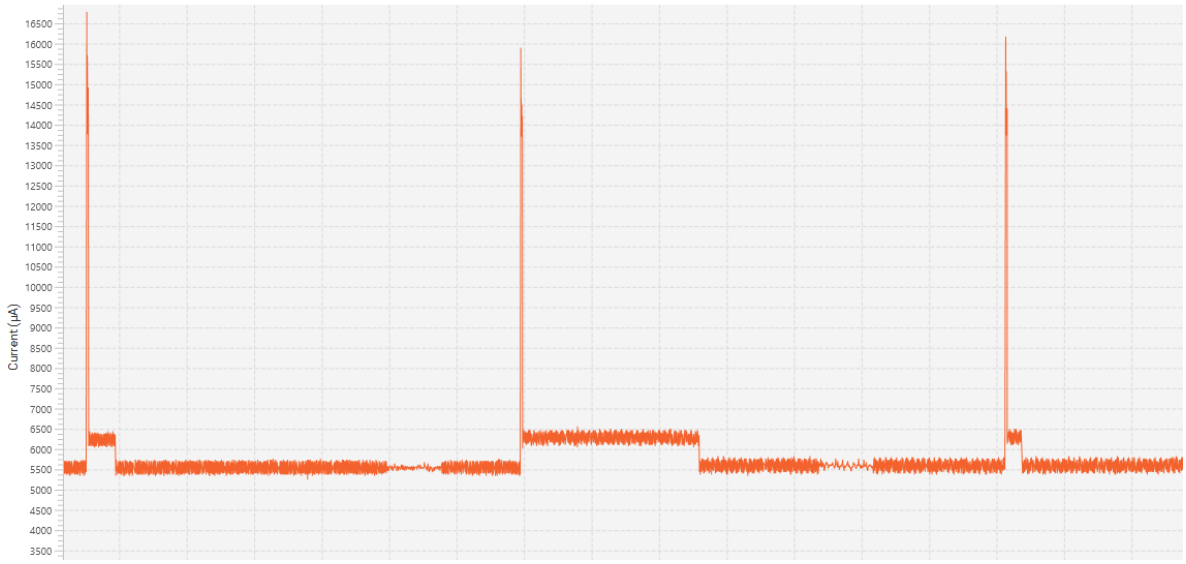


Figure 42: Sleep mode with wfi and wake up by Push Button interrupt (STM32446)

4.2 WFE instruction: Wait For Event

After this instruction, the CPU will enter the Sleep mode conditionally. When wfe is used, the content of the event register is checked:

- If the event register is 1, it resets it to 0 and the processor keeps running.
- If the event register is 0, it goes to Sleep mode.

Software cannot read or write the event register.

4.2.1 What is an event?

In the case of an interrupt, when a peripheral raises its specific flag, the processor executes the corresponding Interrupt routine.

In the case of an event, when a peripheral raises its specific flag, the processor can be aware that the event has taken place (wake up for example) but it does not launch an interrupt routine.

Only a few peripheral has an event register, but all peripherals can generate events.

4.2.2 How to generate events in a STM32

In the block diagram of the Figure 43, we can see that there are two main streams, the interrupt (1), and the events (2):

- The interrupts go to the NVIC.
- The events go to the cortex M.

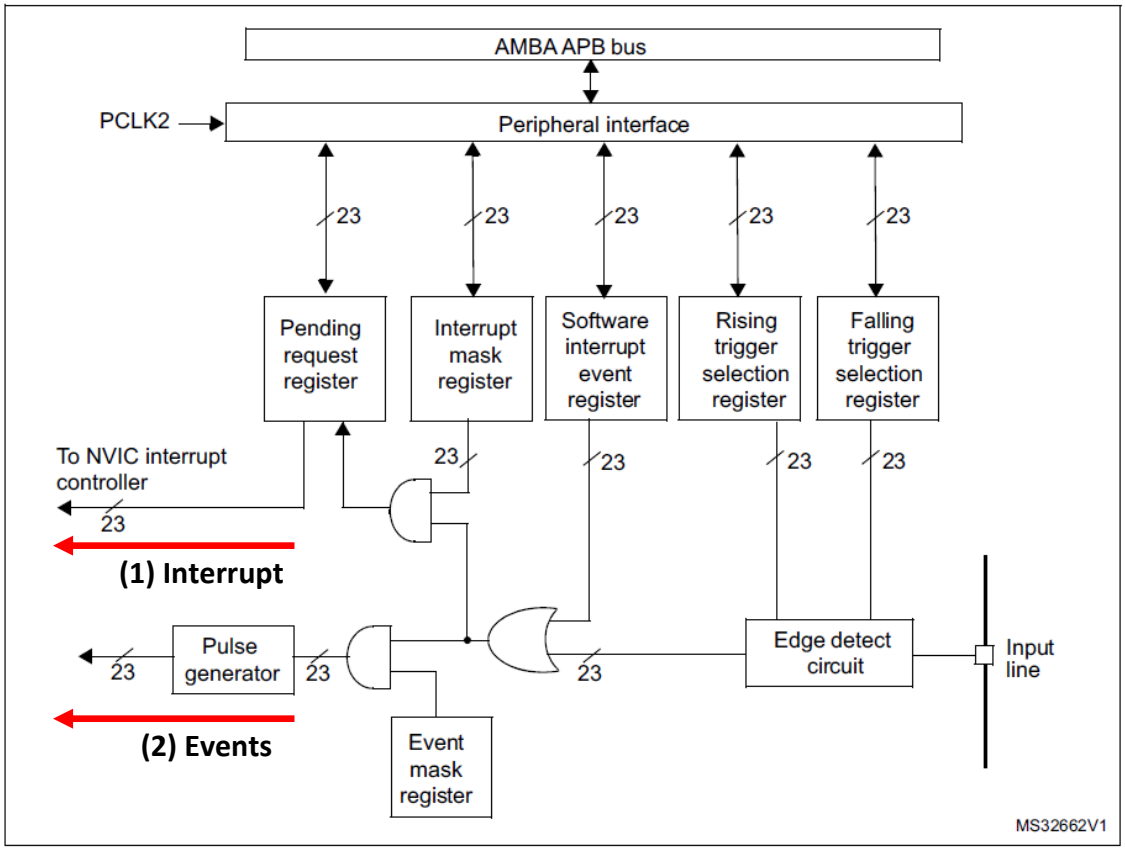


Figure 43: Interrupt/Event controller block diagram

The 23 lines correspond to the 23 EXTI lines: 15 for the GPIO and 7 other peripherals (USB, RTC…). These 23 EXTI lines can create either IT or Events and each of these 23 lines are coming from any GPIO:

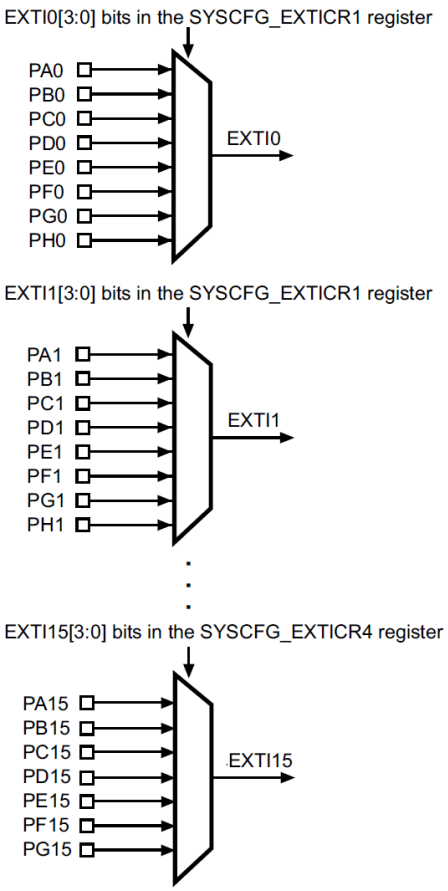


Figure 44: Mapping of the first 15 EXTI lines- Reference Manual STM32F446

The seven other EXTI lines are connected as follows:

- EXTI line 16 is connected to the PVD output
- EXTI line 17 is connected to the RTC Alarm event
- EXTI line 18 is connected to the USB OTG FS Wakeup event
- EXTI line 20 is connected to the USB OTG HS (configured in FS) Wakeup event
- EXTI line 21 is connected to the RTC Tamper and TimeStamp events
- EXTI line 22 is connected to the RTC Wakeup event

Figure 45: Mapping of the next seven EXTI lines- Reference Manual STM32F446

In our application, we are using the Push Button connected to PC13, which is linked to the EXT13 line. EXT13 can create either an interrupt or an event, depending on the **Interrupt Mask Register** (for the interrupt Line) or on the **Event Mask Register** (for the Event Line).

4.2.3 How to wake up the CPU when we use WFE

The reference manual explains three methods to wake up the CPU when wfe is used. We are going to explore the three of them. One of this method uses the SEVONPEND bit (**S**end **EV**ent **ON PEND**ing bit).

- If the WFE instruction was used to enter the low power mode, the MCU exits the mode as soon as an event occurs. The wakeup event can by generated either by:
  - NVIC IRQ interrupt

    - When SEVEONPEND=0 in the Cortex®-M4 System Control register.

    By enabling an interrupt in the peripheral control register and in the NVIC. When the MCU resumes from WFE, the peripheral interrupt pending bit and the NVIC peripheral IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be cleared.

    Only NVIC interrupts with sufficient priority will wakeup and interrupt the MCU.

    - When SEVEONPEND=1 in the Cortex®-M4 System Control register.

    By enabling an interrupt in the peripheral control register and optionally in the NVIC. When the MCU resumes from WFE, the peripheral interrupt pending bit and (when enabled) the NVIC peripheral IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be cleared.

    All NVIC interrupts will wakeup the MCU, even the disabled ones.

    Only enabled NVIC interrupts with sufficient priority will wakeup and interrupt the MCU.

    - Event

    Configuring a EXTI line in event mode. When the CPU resumes from WFE, it is not necessary to clear the EXTI peripheral interrupt pending bit or the NVIC IRQ channel pending bit as the pending bits corresponding to the event line is not set. It may be necessary to clear the interrupt flag in the peripheral.

Figure 46: The three ways to wake up the CPU when wfe is used – Reference Manual STM32F446

4.2.4 Application: First Wake up possibility: On interrupts

This first wake up method refers to that part of the documentation (from Figure 46).

> - When SEVEONPEND=0 in the Cortex®-M4 System Control register.
>
> By enabling an interrupt in the peripheral control register and in the NVIC. When the MCU resumes from WFE, the peripheral interrupt pending bit and the NVIC peripheral IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be cleared.

This means that any interrupt wakes up the processor and the ISR is executed.

We keep the same application as before (see paragraph 4.1.2), but instead of entering the Low Power mode with wfi, we use wfe. The interrupt of the Push Button PC13 was already configured, so the application should work as before.

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	
	`while(1){`
	`printf("Running the while loop\r\n");`
	`printf("The processor goes to Normal sleep using wfe\r\n\r\n");`
	`HAL_SuspendTick();`
	`HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFE);`
	`}`

Push Button ISR	`void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){`
	`HAL_ResumeTick();`
	`printf("Wake Up by Push Button IT\r\n");`
	`}`

What do we see?

This application is exactly the same as before.

4.2.5 Application: Second Wake up possibility: On EXTI Events

We will see here the second possibility to wake up from low Power mode using wfe. It refers to that part of the documentation (from Figure 46).

> - Event
>
> Configuring a EXTI line in event mode. When the CPU resumes from WFE, it is not necessary to clear the EXTI peripheral interrupt pending bit or the NVIC IRQ channel pending bit as the pending bits corresponding to the event line is not set. It may be necessary to clear the interrupt flag in the peripheral.

As we have seen, EXTI is a processor module that has the ability to send events. We will use the Push Button PC13 to create the event. For that, we need to configure the PC13 pin as an "External Event Mode with Falling edge trigger detection" in CubeMX.

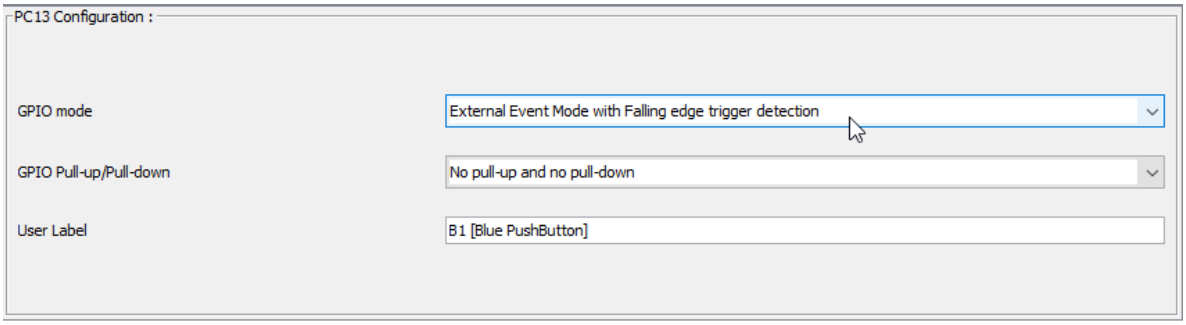


Figure 47: The PC13 pin configuration for generating Event on falling edge

The processor wakes up with the Push Button but no ISR is launched: The PC13 is no longer an interrupt source: it is an event source.

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	
	`while(1){`
	`printf("Running the while loop\r\n");`
	`printf("The processor goes to Normal sleep using wfe\r\n\r\n");`
	`HAL_SuspendTick();`
	`HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFE);`
	`HAL_ResumeTick();`
	`printf("Wake-up by Push Button EXTI event\r\n");`
	`}`

What do we see?

This application no longer executes an interrupt function. It just carries on its execution after the instruction wfe where the MCU has been sent to Sleep Mode.

4.2.6 Application 3: Third Wake up possibility. On pending interrupts

We will see here the third possibility to wake up from low Power mode using wfe. It refers to that part of the documentation (from Figure 46).

> - When SEVEONPEND=1 in the Cortex®-M4 System Control register.
>
> By enabling an interrupt in the peripheral control register and optionally in the NVIC. When the MCU resumes from WFE, the peripheral interrupt pending bit and (when enabled) the NVIC peripheral IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be cleared.
>
> All NVIC interrupts will wakeup the MCU, even the disabled ones.
>
> Only enabled NVIC interrupts with sufficient priority will wakeup and interrupt the MCU.

First, we need to explain properly how interrupt work in a Cortex M processor: The interrupts are controlled by the NVIC. This NVIC controller accepts (if the IT source is unmasked), or not (if the IT source is masked) the interruption. When an IT occurs, a flag is raised. We call this situation a pending IT. If the NVIC accepts it, the Cortex M will be interrupted, and the flag need to be reset.

To enable an interrupt, we have to:

1. Configure the Peripheral to generate the IRQ [ie: **__HAL_UART_ENABLE_IT(huart, UART_IT_TXE**) for generating IT at the end of an USART transmission]
2. Configure the NVIC to accept (unmask) the IT of a specific peripheral [ie: **HAL_NVIC_EnableIRQ(USART2_IRQn)** for unmasking the USART2 interrupt]

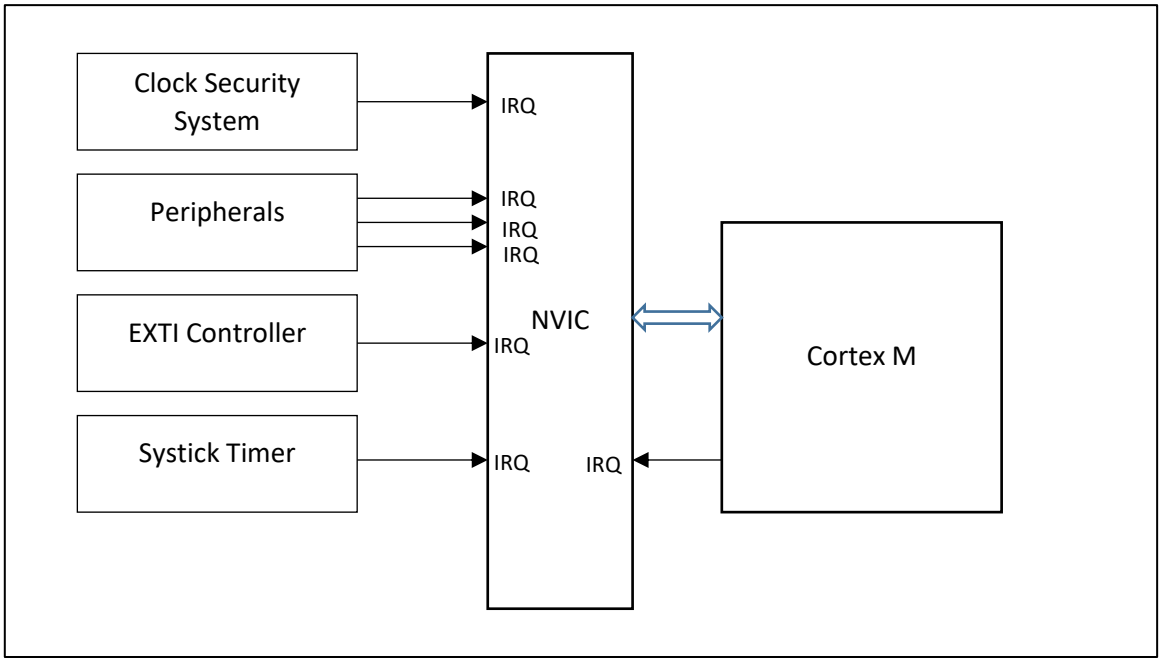


Figure 48 Generation of IRQ in ARM cortex M

What do we want to do?

We want the peripheral to send an event to the MCU. But the peripheral cannot create events by itself. The documentation explains us that we can make it happen by using a special bit called SEVONPEND (**S**end **EV**ent **ON PEND**ing)

1. The SEVONPEND bit must be set: The following HAL function set the SEVONPEND bit.

```
HAL_PWR_EnableSEVOnPend()
```

2. There must be an IT pending bit: PC13 must be able to generate IT.

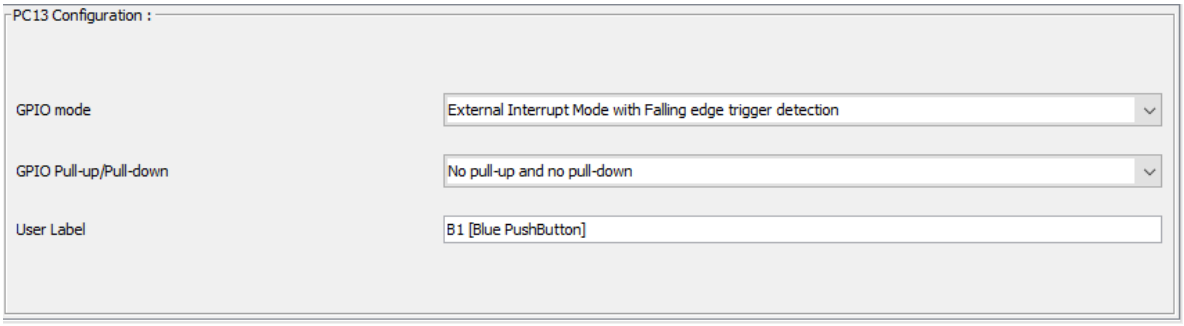


Figure 49: PC13 GPIO mode configuration for generating IT.

3. The corresponding IT shall be disabled in the NVIC, so that IT will be pending but no interrupt will be triggered.

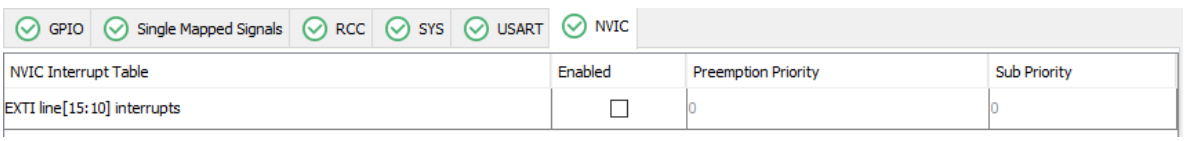


Figure 50: The IT generated by PC13 is unmasked in the NVIC.

4. The documentation also says that when the MCU wakes up, "*the peripheral interrupt pending bit and the NVIC peripheral IRQ pendant bit have to be cleared*".

- Clear the peripheral IT flag: **__HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13)**
- Clear the NVIC IT pending flag: **HAL_NVIC_ClearPendingIRQ(EXTI15_10_IRQn)**

The code of our application is as follow:

Function	Code

main( )	`HAL_PWR_EnableSEVOnPend();`
	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	
	`While(1){`
	`printf("Running the while loop\r\n");`
	`printf("The processor goes to Normal sleep using wfe\r\n\r\n");`
	`HAL_SuspendTick();`
	`__HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);`
	`HAL_NVIC_ClearPendingIRQ(EXTI15_10_IRQn);`
	`HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFE);`
	`HAL_ResumeTick();`
	`printf("Wake up by Push Button pending IT");`
	`}`

Note: __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13) and HAL_NVIC_ClearPendingIRQ (EXTI15_10_IRQn) have to be placed just before the HAL_PWR_EnterSLEEPMode. Otherwise, if the MCU goes to sleep with one of these two flags already ON, it will not wake up.

4.3 When to use WFI or WFE?

Most of the time we can use wfi. When we use wfe we don't launch an ISR, so there is no context switching (stacking / unstacking). Therefore, it is easier to implement and faster to respond.

We want to show by an application the difference between the **wfi** (interrupt) and the **wfe** (event) instructions on the speed point of view. A push button is going to wake up the processor and we measure the time between the Push Button signal (falling edge) and the action realized by the processor when it wakes up. In our case, it would be a simple led turning on PA5. For that purpose, we use a logic analyser or an oscilloscope.

Consumption values and wake up time are stored in the table below.

Test Conditions	MCU	Low Power Mode	Wake up time
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	Normal Sleep Mode with **wfi**	1.8 µs
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	STM32F446	Normal Sleep Mode with **wfe**	780 ns

Table 20: Wake up time depending on the way we entered the Sleep mode

In this application, we are still using the Sleep mode. Here is the power consumption during sleep time:

Test Conditions	Low Power Mode	Current consumption during Low Power
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	Normal sleep mode	5.85 mA

5 The power domains

STM32 have separated power supplies, which have different purposes. The segmentation of the power scheme is useful for controlling the MCU consumption. In this chapter, we will have a look on the power domains of the MCU.

5.1 Power supply overview

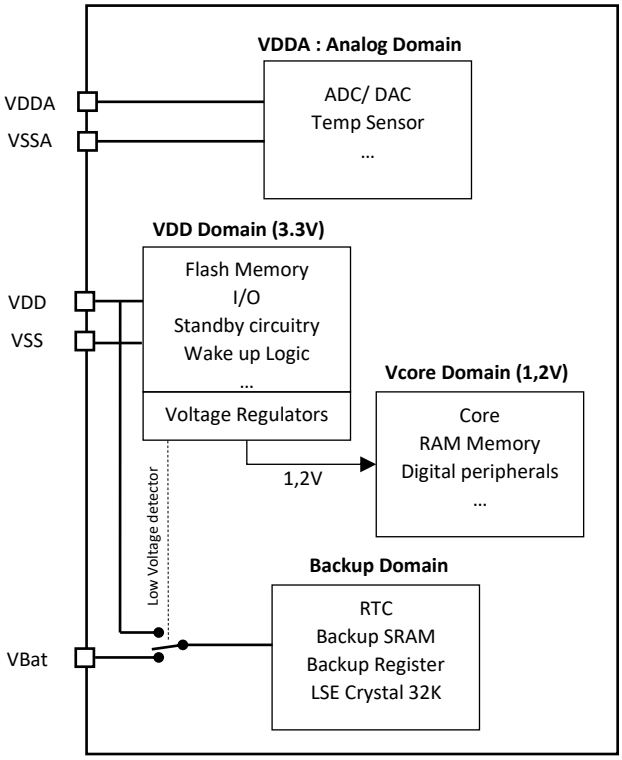


Figure 51: Simplified Power supply overview of the STM32F446

5.2 The regulators

For delivering the Vcore voltage (1.2V), two regulators can be used.

- The Main Regulator (MR)
- The Low Power Regulator (LPR)

Using the LPR, obviously reduces the consumption but it also increases the wake up time. LPR is not always available.

- Running mode: Only Main Regulator is available.
- Normal Sleep mode: Only Main Regulator is available.
- Stop mode: Main Regulator **OR** Low Power Regulator available.
- Stanby mode: No regulator available.

5.3 The regulators modes

What is a bit complex is that each regulator (MR and LPR) can use five different configurations. All configuration are only available as specified on the Figure 52.

Voltage regulator configuration	Run mode	Sleep mode	Stop mode	Standby mode
Normal mode	MR (1)	MR (1)	MR or LPR (2)	-
Low-voltage mode	-	-	MR or LPR	-
Over-drive mode(2)	MR (1)	MR (1)	-	-
Under-drive mode	-	-	MR or LPR (2)	-
Power-down mode	-	-	-	Yes (3)

Figure 52: Voltage regulator mode versus operating mode

5.3.1 Case (1) - Figure 52

We are going to explain the case (1) on the Figure 52. When the processor is running (Run mode) or in Sleep mode, the MR is the only regulator available. The MR regulators provide full power to the 1.2V domain. The exact value of the 1.2V can be scaled (level 1, 2 or 3) in order to adjust the power delivered and reach the maximum frequency. The "over drive mode" is made for overclocking the MCU up to 180 MHz, but obviously consumes more current.

Power Scale 3 ((VOS[1:0] bits in PWR_CR register = 0x01), 120 MHz HCLK max frequency	1.08	1.14	1.20	V
Power Scale 2 ((VOS[1:0] bits in PWR_CR register = 0x10), 144 MHz HCLK max frequency with over-drive OFF or 168 MHz with over-drive ON	1.20	1.26	1.32	
Power Scale 1 ((VOS[1:0] bits in PWR_CR register = 0x11), 168 MHz HCLK max frequency with over-drive OFF or 180 MHz with over-drive ON	1.26	1.32	1.40	

Figure 53: Voltage level for the Main Regulator

The Figure 54 shows the HCLK maximum frequency:

- Scale 3 for HCLK < 120 MHz
- Scale 2 for 120 MHz < HCLK <144 MHz (normal mode) / 168 MHz (over-drive mode)
- Scale 1 for 144 MHz < HCLK < 168 MHz (normal mode) / 180 MHz (over-drive mode)

Symbol	Parameter	Conditions[1]		Min	Typ	Max	Unit
f_{HCLK}	Internal AHB clock frequency	Power Scale 3 (VOS[1:0] bits in PWR_CR register = 0x01), Regulator ON, over-drive OFF		0	-	120	MHz
		Power Scale 2 (VOS[1:0] bits in PWR_CR register = 0x10), Regulator ON	Over-drive OFF	0	-	144	
			Over-drive ON		-	168	
		Power Scale 1 (VOS[1:0] bits in PWR_CR register= 0x11), Regulator ON	Over-drive OFF	0	-	168	
			Over-drive ON		-	180	

Figure 54: HCLK frequency possible with the power scale 1, 2 or 3

In CubeMX, everything is properly configured as soon as you change the frequency of HCLK. You can check the scale number and the overdrive mode in **CubeMX > System Core > RCC > Parameters Settings > Power Parameters.**

- Try to change the frequency of HCLK and verify that the value generated for the scale number and overdrive mode are correct.

5.3.2 Case (2) - Figure 52

When the MCU is in Stop mode, we have the choice between the MR and LPR. The latter will obviously reduce the power consumption but the wake up time will increase. If we use the regulators in underdrive mode, their leakage current will be reduce but the Flash Memory is not powered anymore. Once again, the wake-up time will increase.

5.3.3 The regulators (MR or LPR) in "power down mode"

The power down mode is automatically activated when the CPU is in Standby Mode: SRAM, RTC, Registers are not powered anymore.

6 Exploring the Stop Mode

6.1 Entering the Stop Mode

6.1.1 Choosing the regulator and its configuration

As we have seen in the previous chapter, in Stop mode, we can choose between LPR (Low Power Regulator) instead of the MR (Main Regulator). The benefits will be a lower power consumption, but there will be a higher wake up time. As we can see in Figure 55, the PPDS bit can select the right regulator in Stop mode.

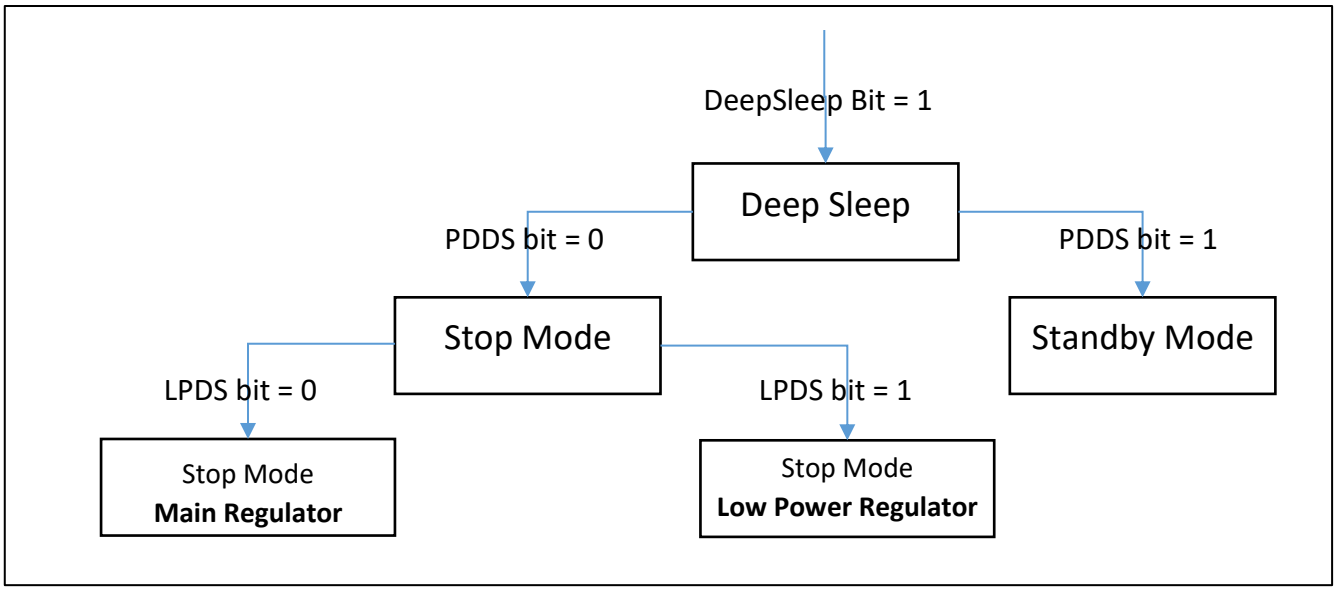


Figure 55: The two Stop modes in a STM32F446

How do we enter Stop mode?

- Select the Deep Sleep mode (Cortex ARM specific)
- Set PDDS bit = 0. (STM32 specific)
- Select the regulator we want to use: PWR_MAINREGULATOR_ON if we want the MR ON, or LWR_LOWPOWERREGULATOR_ON if we want the LPR ON.

To enable the underdrive in Low power **stop mode**, we use the Power Control Register:

- bit UDEN, in order to activate the under drive mode capability

Then we select the regulator concerned by the underdrive mode:

- bit MRUDS for the under drive on the MR
- bit LPUDS for the under drive on the LPR

Everything is done by the HAL function **HAL_PWR_EnterSTOPMode()** for the Normal mode or **HAL_PWREx_EnterUnderDriveSTOPMode()** for underdrive mode.

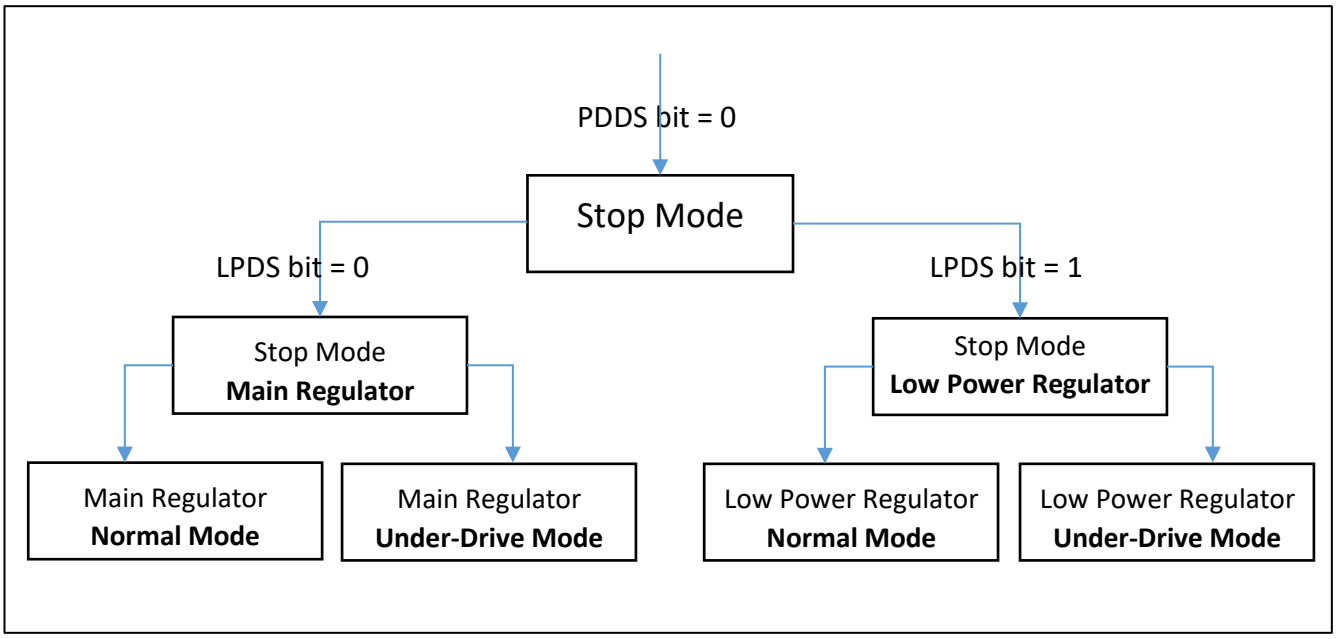


Figure 56: Underdrive mode for the MR and the LPR

6.1.2 Flash memory: ON or OFF

When using the underdrive mode, we saw that the Flash memory is always off. However, if we choose the normal mode for MR or LPR, then we can choose whether we want the Flash memory ON or OFF.

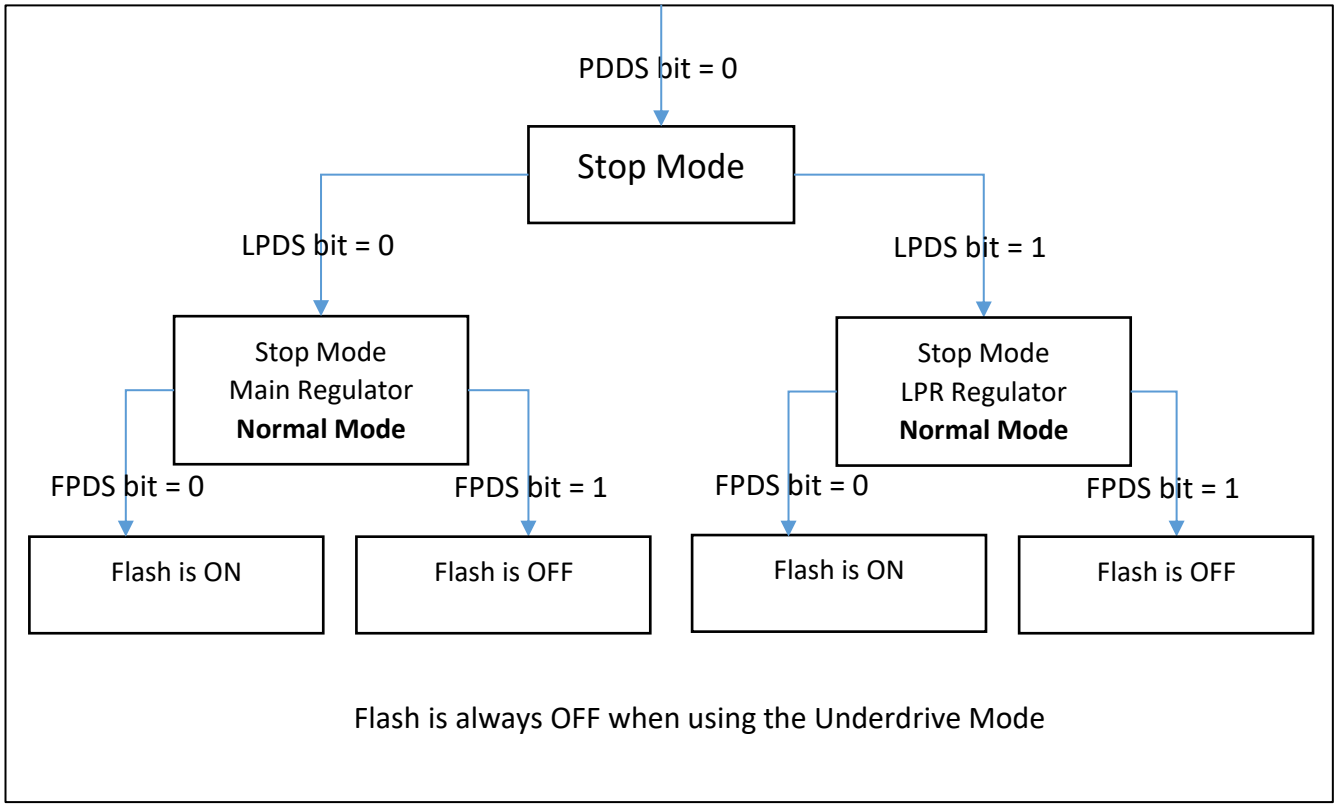


Figure 57: Selection of the flash memory ON or OFF during STOP mode.

6.2 Test of the Stop Modes

When exiting the Stop mode, the STM32 use the HSI RC oscillator with its default configuration. Therefore, we have to reconfigure the clock system each time we exit the Stop mode. In our application, if we don't do it, the USART will not work.

We are going to use the application from the chapter 4.1.2: The push button generates an interrupt that wakes up the MCU. For each test, we will change the configuration (MR, LPR, underdrive, Flash memory) and measure the power consumption and the wake up time. We will test the six following configurations:

1. MR in Normal Mode + Flash ON

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	
	`While(1){`
	`printf("Running the while loop\r\n");`
	`printf("The processor goes to Normal sleep using wfi\r\n\r\n");`
	`HAL_PWR_EnterSTOPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);`
	`}`

Push Button ISR	`void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){`
	`SystemClock_Config();`
	`printf("Wake Up by Push Button IT\r\n");`
	`}`

2. MR in Normal Mode + Flash OFF

```
HAL_PWREx_EnableFlashPowerDown();
HAL_PWR_EnterSTOPMode(PWR_MAINREGULATOR_ON, PWR_STOPENTRY_WFI);
```

3. LPR ON in Normal Mode + Flash ON

```
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
```

4. LPR ON in Normal Mode + Flash OFF

```
HAL_PWREx_EnableFlashPowerDown();
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
```

5. MR in Under Drive Mode (Flash is always OFF)

```
HAL_PWREx_EnableFlashPowerDown();
HAL_PWREx_EnterUnderDriveSTOPMode(PWR_MAINREGULATOR_UNDERDRIVE_ON,
PWR_STOPENTRY_WFI)
```

6. LPR in Under Drive Mode (Flash is always OFF)

```
HAL_PWREx_EnableFlashPowerDown();
HAL_PWREx_EnterUnderDriveSTOPMode(PWR_LOWPOWERREGULATOR_UNDERDRIVE_ON,
PWR_STOPENTRY_WFI)
```

Consumption values and wake up time are stored in the chart and table below.

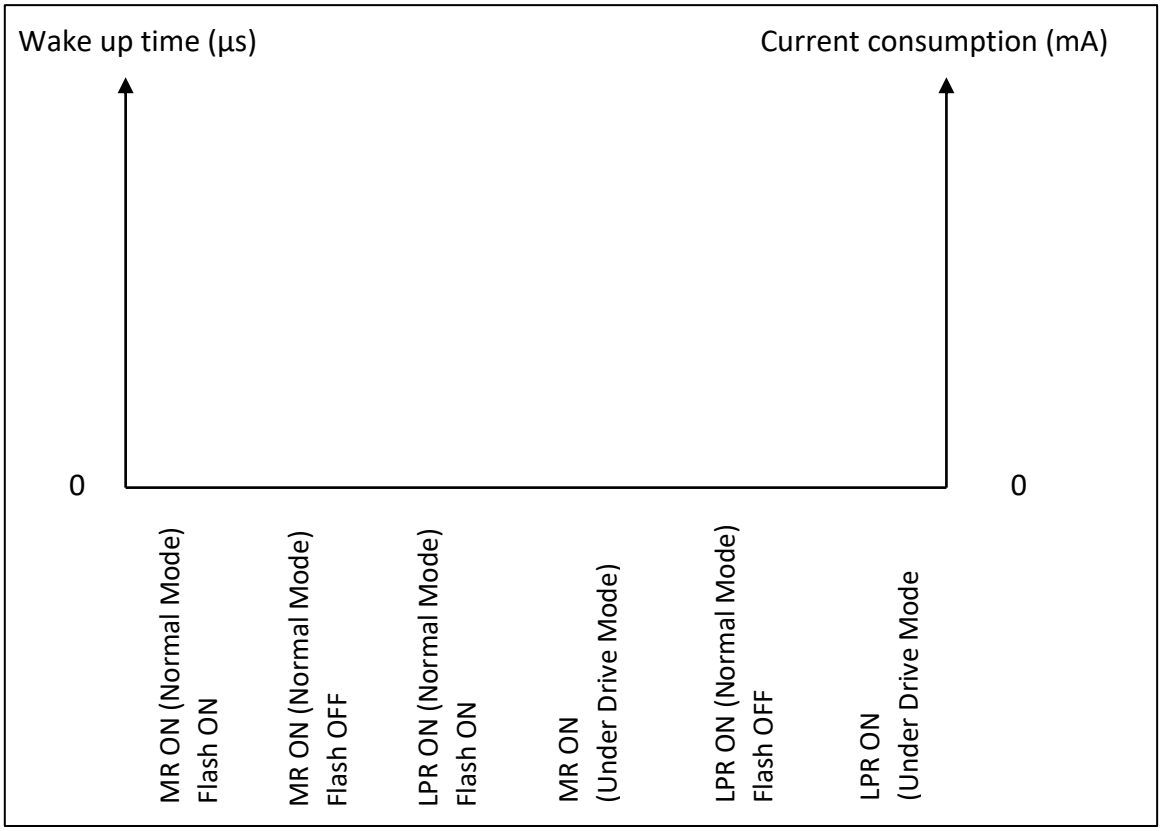


Figure 58: Current consumption and wake up time in different STOP low power mode

Test Conditions	Low Power Mode	Current consumption	Wake up time
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	MR ON (Normal Mode) Flash ON	761 µA	21.2 µs
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	MR ON (Normal Mode) Flash OFF	732 µA	112.9 µs
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	LPR ON (Normal Mode) Flash ON	676 µA	23.2 µs
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	MR ON (Under Drive Mode) Flash OFF	678 µA	119.8 µs
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	LPR ON (Normal Mode) Flash OFF	647 µA	122 µs
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI)	LPR ON (Under Drive Mode) Flash OFF	573 µA	122,7 µs

Table 21: Current consumption and wake up time in stop mode with the STM32F446 MCU

You can get useful information with the Table 22 for the configuration of the STOP mode.

Voltage Regulator Mode		UDEN[1:0] bits	MRUDS bit	LPUDS bit	LPDS bit	FPDS bit	Wakeup latency
Normal mode	STOP MR (Main Regulator)	-	0	-	0	0	HSI RC startup time
	STOP MR- FPD	-	0	-	0	1	HSI RC startup time + Flash wakeup time from power-down mode
	STOP LP	-	0	0	1	0	HSI RC startup time + regulator wakeup time from LP mode
	STOP LP-FPD	-	-	0	1	1	HSI RC startup time + Flash wakeup time from power-down mode + regulator wakeup time from LP mode
Under-drive Mode	STOP UMR-FPD	3	1	-	0	-	HSI RC startup time + Flash wakeup time from power-down mode + Main regulator wakeup time from under-drive mode + Core logic to nominal mode
	STOP ULP-FPD	3	-	1	1	-	HSI RC startup time + Flash wakeup time from power-down mode + regulator wakeup time from LP under-drive mode + Core logic to nominal mode

Table 22: Configuration of the STOP low power mode – Reference Manual STM32F446

- We can also compare the result with the Table 23 values from the Reference Manual (STM32F446).

Symbol	Parameter	Conditions	Typ	Max			Unit
				V_{DD} = 3.6 V			
			T_A = 25 °C	T_A = 25 °C(1)	T_A = 85 °C	T_A = 105 °C(1)	
$I_{DD\_STOP\_NM}$ (normal mode)	Supply current in Stop mode with voltage regulator in main regulator mode	Flash memory in Stop mode, all oscillators OFF, no independent watchdog	0.234	1.2	10	16	mA
		Flash memory in Deep power down mode, all oscillators OFF, no independent watchdog	0.205	1	9.5	15	
	Supply current in Stop mode with voltage regulator in Low Power regulator mode	Flash memory in Stop mode, all oscillators OFF, no independent watchdog	0.15	0.95	8.5	14	
		Flash memory in Deep power down mode, all oscillators OFF, no independent watchdog	0.121	0.9	6	12	
$I_{DD\_STOP\_UDM}$(under-drive mode)	Supply current in Stop mode with voltage regulator in main regulator and under-drive mode	Flash memory in Deep power down mode, main regulator in under-drive mode, all oscillators OFF, no independent watchdog	0.119	0.4	3	5	
	Supply current in Stop mode with voltage regulator in Low Power regulator and under-drive mode	Flash memory in Deep power down mode, Low Power regulator in under-drive mode, all oscillators OFF, no independent watchdog	0.055	0.35	3	5	

Table 23: Typical and maximum current consumption in STOP modes

7 Exploring the Standby Mode

This Low power mode puts the MCU in Deep Sleep mode with the lowest power consumption possibly achieved. The MCU will switch off all clocks, Flash memory, regulators, SRAM memory and registers. All values are lost, except the ones stored in the register backup domain, and the backup SRAM.

7.1 The Standby mode

7.1.1 New application

For this chapter, our application will enter the standby mode if the user push button is pressed (PC13) and will be woken up on a rising edge on the GPIO PA0.

7.1.2 : Entering and exiting the standby mode

Two PINs can wake up the MCU: WKUP1 or WKUP2. Before entering the Standby mode, we need to enable them. A rising edge on the Wake up pin of the MCU wakes it up from Standby mode.

- The first wake up pin is PA0 (WKUP0 in the datasheet, called WKUP1 in the Reference Manual)
- The second wake up pin is PC13 (WKUP1 in the datasheet, called WKUP2 in the Reference Manual)

While programming, we need to keep the designation of the Reference Manual. The register **PWR_CSR -> EWUP1** programs the Wake up pin WKUP1 (PA0) and **PWR->EWUP2** programs the wake up pin WKUP2 (PC13). We can use the HAL function HAL_PWR_EnableWakeUpPin().

When a rising edge is applied on the Wake up pin, the flag PWR_CSR -> **WUF** (**W**ake **U**p **F**lag) is set and it wakes up the MCU. This Flag has to be cleared by software using the HAL macro __HAL_PWR_CLEAR_FLAG(**PWR_FLAG_WU**), otherwise, the CPU will wake up continuously.

Here is the code of our new application.

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	`HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);`
	
	`while (1){`
	`printf("Waiting for the Push Button PC13...\r\n");`
	`while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)==SET);`
	`printf("The user pressed the PC13 Push Button\r\n");`
	`printf("The processor goes to Standby mode\r\n\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);`
	`HAL_PWR_EnterSTANDBYMode();`
	`}`

The wake up pin resets the MCU, so we never go beyond the HAL_PWR_EnterSTANDBYMode() function.

Test Conditions	Low Power Mode	Current consumption
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **115200**	Sleep mode	5,50 mA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **115200**	Stop mode	732 µA
Default Mode (CubeMX) with HCLK = **84 MHz** (HSI) BaudRate = **115200**	Standby mode	3 µA

Table 24: Current consumption of the STM32F446 depending the Low Power mode

7.1.3 Knowing the previous state of the MCU

In our application, we don't know if we have previously been in Standby mode or if the application is running for the first time. If the MCU is waking up from Standby mode, you might not want to execute the same instructions. The **S**tand **B**y **F**lag (SBF) of the Power Control Register is set as soon as the processor is going to Standby mode, so we can check it at the beginning of our application.

- The HAL macro **__HAL_PWR_GET_FLAG**(PWR_FLAG_SB) checks the flag SB flag value.
- The HAL macro **__HAL_PWR_CLEAR_FLAG**(PWR_FLAG_SB) resets the SB flag.

We are going to improve the previous application to check if the processor was previously in Standby mode.

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	`HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);`
	
	`if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB) == 1){`
	`printf("The MCU was in Standby mode\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);`
	`}`
	
	`while (1){`
	`printf("Waiting for the Push Button PC13...\r\n");`
	`while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)==SET);`
	`printf("The user pressed the PC13 Push Button\r\n");`
	`printf("The processor goes to Standby mode\r\n\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);`
	`HAL_PWR_EnterSTANDBYMode();`
	`}`

What do we see?

- If we put the STM32 in Standby mode with the user push button PC13, a rising edge on PA0, or a reset (black push button) will reset the MCU. The application will state that the MCU was in standby mode.
- If we don't put the STM32 in Standby mode, a rising edge an PA0 has no effect. A reset (black push button) will restart the application.

7.1.4 Differencing the system Reset and the Wake up pin Reset

In our previous application, when the STM32 goes to Standby mode, we cannot make the difference between System Reset (the application runs for the first time) and a wake up from the WKUP pin

(the application wakes up). If we want this information, we need to verify the **W**ake **U**p **F**lag (WUF) of the Power Control Register.

We update our application with the following code:

Function	Code

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	`HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);`
	
	`if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB) == 1){`
	`printf("The MCU was in Standby mode\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);`
	`}`
	
	`if(__HAL_PWR_GET_FLAG(PWR_FLAG_WU) == 1){`
	`printf("The user pressed the WKUP PIN\r\n");`
	`}`
	`else{`
	`printf("The user pressed the RESET PIN\r\n");`
	`}`
	
	`while (1){`
	`printf("Waiting for the Push Button PC13...\r\n");`
	`while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)==SET);`
	`printf("The user pressed the PC13 Push Button\r\n");`
	`printf("The processor goes to Standby mode\r\n\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);`
	`HAL_PWR_EnterSTANDBYMode();`
	`}`

7.2 The backup domain

The backup domain is the only part powered via V_{BAT} pin during standby mode. It includes:

- 4 Ko of backup SRAM
- 20 backup registers
- The Real Time Clock (RTC)

7.2.1 The backup SRAM

In our application, when we reset the processor, we lose all data stored in RAM memory. However, in many applications we need to keep the value of variables.

The backup SRAM is an EEPROM-like memory area. It can be used to store data that need to be retained while the processor is in Standby mode. The backup SRAM is disabled by default but it can be enabled by software. If we want to keep the data during Standby mode, we need a power source on V_{BAT}. We can check on the Nucleo user guide that the V_{BAT} pin is connected to the VDD power supply.

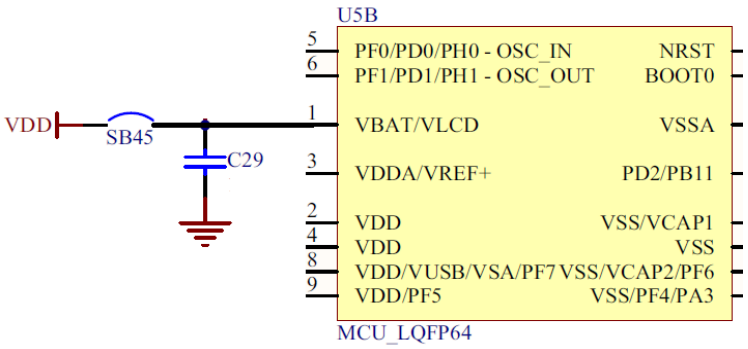


Figure 59: VBAT connected to VDD on the Nucleo board

The SRAM backup domain peripheral is "write protected" by default. The following example gives the procedure to enable the access to the backup SRAM.(Reference Manual):

- Access to the backup SRAM

1. Enable the power interface clock by setting the PWREN bits in the RCC_APB1ENR register (see *Section 6.3.13*).
2. Set the DBP bit in the *PWR power control register (PWR_CR)* to enable access to the backup domain
3. Enable the backup SRAM clock by setting BKPSRAMEN bit in the *RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)*.

Figure 60: How to enable the backup SRAM – Reference Manual STM32F446

```
1. __HAL_RCC_PWR_CLK_ENABLE();          // Enable power interface clock
2. HAL_PWR_EnableBkUpAccess();          // Enable Access to backup domain
3. __HAL_RCC_BKPSRAM_CLK_ENABLE();      // Enable backup SRAM clock
```

The base address of the backup SRAM is given in the datasheet and defined in the include file: **Drivers > CMSIS >Device > ST > STM32F4xx > Include > stm32f446xx.h** with the name BKPSRAM. That is where we are going to write data.

```
uint32_t* pBackupVariable = (uint32_t*) BKPSRAM_BASE;
```

Now that we have located the backup SRAM, we need to keep the value while being in Standby mode. Therefore, we need to activate the backup voltage regulator: HAL_PWREx_EnableBkUpReg()

We use the following code:

Function	Code
Global variables	`uint32_t* pBackupVariable = (uint32_t*) BKPSRAM_BASE;` `uint32_t  randomVariable;`

main( )	`// Enable Backup SRAM Access`
	`__HAL_RCC_PWR_CLK_ENABLE();`
	`HAL_PWR_EnableBkUpAccess();`
	`__HAL_RCC_BKPSRAM_CLK_ENABLE();`
	
	`// Enable BackUp SRAM regulator`
	`HAL_PWREx_EnableBkUpReg();`
	
	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	`HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);`
	
	`if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB) == 1){`
	`printf("The MCU was in Standby mode\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);`
	`}`
	
	`if(__HAL_PWR_GET_FLAG(PWR_FLAG_WU) == 1){`
	`printf("The user pressed the WKUP PIN\r\n");`
	`}`
	`else{`
	`printf("The user pressed the RESET PIN\r\n");`
	`}`
	
	`printf("randomVariable : %X\n\r",randomVariable);`
	`printf("backupVariable : %X\n\r",*pBackupVariable);`
	
	`while(1)`
	`{`
	`printf("Waiting for the Push Button PC13...\r\n");`
	`while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)==SET);`
	`printf("The user pressed the PC13 Push Button\r\n");`
	`printf("Setting randomVariable to 0xAAAAAAAA\r\n");`
	`printf("Setting pBackupVariable to 0xBBBBBBBB\r\n");`
	`randomVariable= 0xAAAAAAAA;`
	`*pBackupVariable = 0xBBBBBBBB;`
	`printf("The processor goes to Standby mode\r\n\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);`
	`HAL_PWR_EnterSTANDBYMode();`
	`}`

What can we see?

- The backupVariable value is still the same after Standby mode.
- The randomVariable has been reset after Standby mode.

If you remove the **HAL_PWREx_EnableBkUpReg**() function from the application, both variables will be reset after Standby mode.

8 The RTC

8.1 General overview

A RTC gives the time and date in real time. In low power embedded system, the RTC peripheral can keep running even in the lowest low power mode.

8.1.1 The four modules of the RTC

The **R**eal **T**ime **C**lock (**RTC**) embedded in STM32 microcontroller acts as an independent BCD timer, as long as the operating voltage remains ON. It does not stop in low power mode or during Reset. The calendar can gives information on: years, months, days, hours, minutes, seconds and sub-seconds.

- Two alarms can interrupt the MCU on a date.
- Wake up IT and event can occur

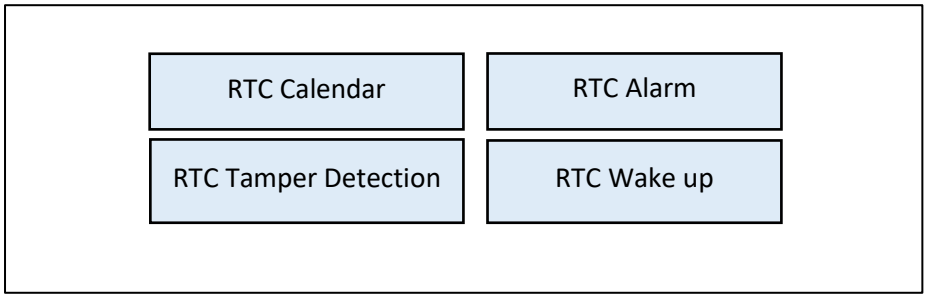


Figure 61: The four RTC Module

8.1.2 The RTC Date and Time Register

- The RTC_**DR** (RTC **D**ate **R**egister) stores the Date
- The RTC_**TR** (RTC **T**ime **R**egister) stores the Time

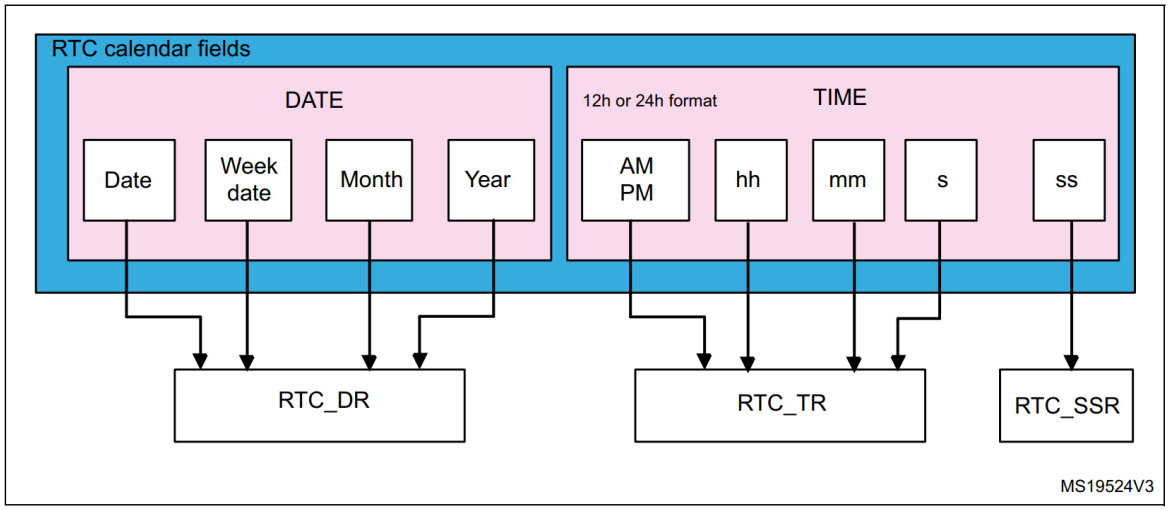


Figure 62: The RTC calendar field – Application Note 4754

8.1.3 The clock source

There are three different clock sources available for the RTC. We can see them on the Figure 63:

- LSE (32768 kHz)
- HSE_RTC
- LSI

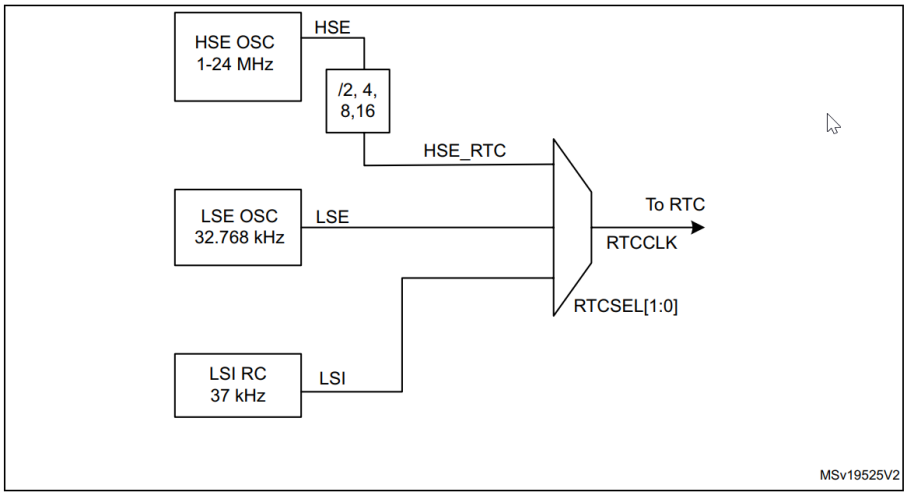
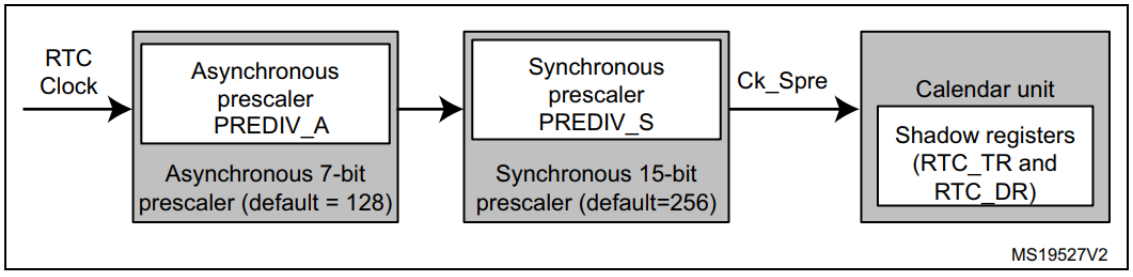


Figure 63: The RTC Clock Source – Application Note 4754

The RTC clock frequency has to be 1 Hz. Therefore, we use two PREDIV (A and S) to generate the proper rate on ck_spre as we can see in the Figure 65 and Figure 64.



The formula to calculate ck_spre is:

$$ck\_spre = \frac{RTCCLK}{(PREDIV\_A + 1) \times (PREDIV\_S + 1)}$$

Figure 64 : Formula to calculate the clock - Application Note 4759

RTCCLK Clock source	Prescalers		ck_spre
	PREDIV_A[6:0]	PREDIV_S[14:0]	
HSE_RTC = 1 MHz	124 (div 125)	7999 (div 8000)	1 Hz
LSE = 32.768 kHz	127 (div 128)	255 (div 256)	1 Hz
LSI = 32 kHz	127 (div 128)	249 (div 250)	1 Hz
LSI = 37 kHz	124 (div 125)	295 (div 296)	1 Hz
LSI = 40 kHz	127 (div 128)	311 (div 312)	1 Hz

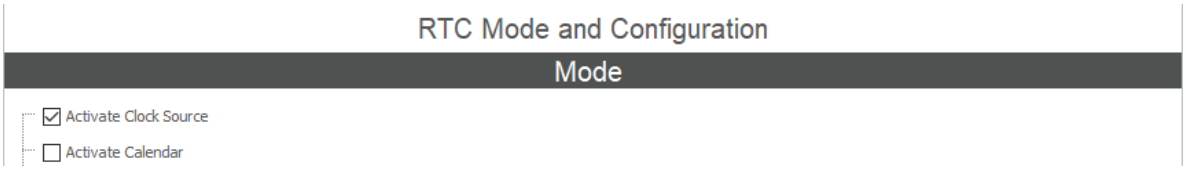
Figure 65: PREDIV A and S values for different clock sources

8.2 Using the RTC

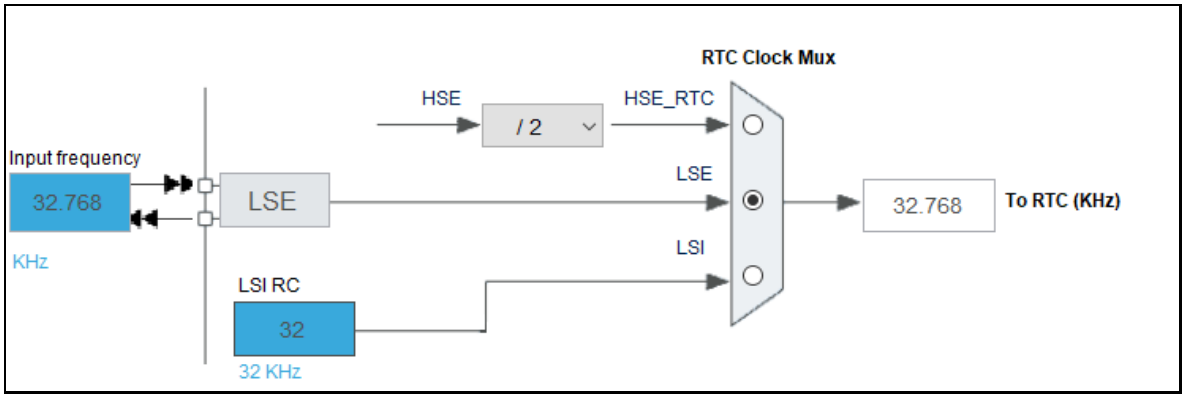
8.2.1 Reading the Date and Time

We want to create an application, which reads the time and date continuously from the RTC.

1. Enable the RTC peripheral in STM32CubeMX



2. Chose the Clock source for the RTC: We will chose the external 32.768 kHz crystal on the Nucleo board.



3. Tune the Prescaler in order to have 1Hz on ck_spre.

$$ck\_spre = \frac{32768}{(127+1)*(255+1)} = 1\,Hz$$

We use the following code for our application:

Function	Code
Global variables	`RTC_TimeTypeDef myTime;` `RTC_DateTypeDef myDate;`

main( )	`printf("\r\n\r\nTest of STM32 RTC\r\n");`
	
	`while (1){`
	`HAL_RTC_GetTime(&hrtc, &myTime, RTC_FORMAT_BIN);`
	`HAL_RTC_GetDate(&hrtc, &myDate, RTC_FORMAT_BIN);`
	
	`printf("Time : %d:%d:%d\r\n",myTime.Hours,myTime.Minutes,myTime.Seconds);`
	`printf("Date : %d/%d/%d\r\n\r\n",myDate.Date,myDate.Month,myDate.Year);`
	`HAL_Delay(1000);`
	`}`

What can we see?

If we reset the CPU, the RTC keeps its value. However, if we remove the power, the time and date will be lost. We will have the same behaviour with the Standby low power mode.

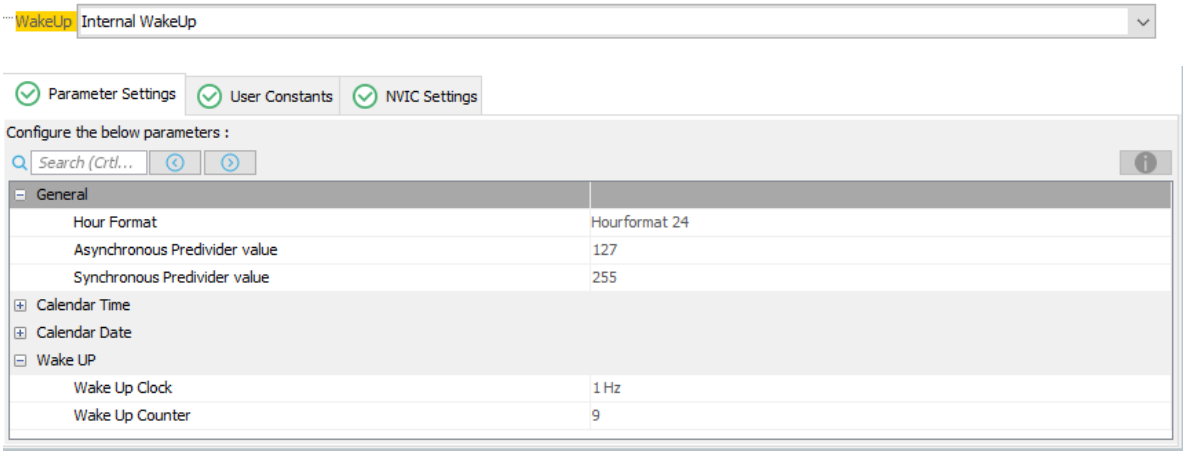
8.2.2 The Wake Up unit

The wake up unit is a down counting timer which can generate an interrupt and wake up the STM32 even in the Standby mode.

Our application:

We will create an application, which goes to Standby mode and wakes up every 10 seconds.

1. Enable and configure the wake up timer in STM32CubeMx.



2. Unmask interruption of the wake up timer:

3. Generate the code
4. In the MX_RTC_Init() function, remove the HAL_RTC_SetTime() and HAL_RTC_SetDate() function to prevent the MCU to erase the actual date and time when the microcontroller restarts.

We use the following code for our application:

Function	Code
Global variables	`RTC_TimeTypeDef myTime;` `RTC_DateTypeDef myDate;`

main( )	`printf("\r\n\r\nTest of Low Power Application on STM32\r\n");`
	
	`if(__HAL_PWR_GET_FLAG(PWR_FLAG_WU) == 1){`
	`printf("The Wake up Timer Restarted the MCU\r\n");`
	`}`
	
	`while (1){`
	`HAL_RTC_GetTime(&hrtc, &myTime, RTC_FORMAT_BIN);`
	`HAL_RTC_GetDate(&hrtc, &myDate, RTC_FORMAT_BIN);`
	
	`printf("Time : %02d:%02d:%02d\r\n",myTime.Hours,myTime.Minutes,myTime.Seconds);`
	`printf("Date : %02d/%02d/%02d\r\n\r\n",myDate.Date,myDate.Month,myDate.Year);`
	`printf("The processor goes to Standby mode\r\n\r\n");`
	`__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);`
	`HAL_PWR_EnterSTANDBYMode();`
	`}`

What can we see?

The STM32 wakes up every 10 seconds. The RTC values is not reset because it is part of the Backup domain.

Appendices

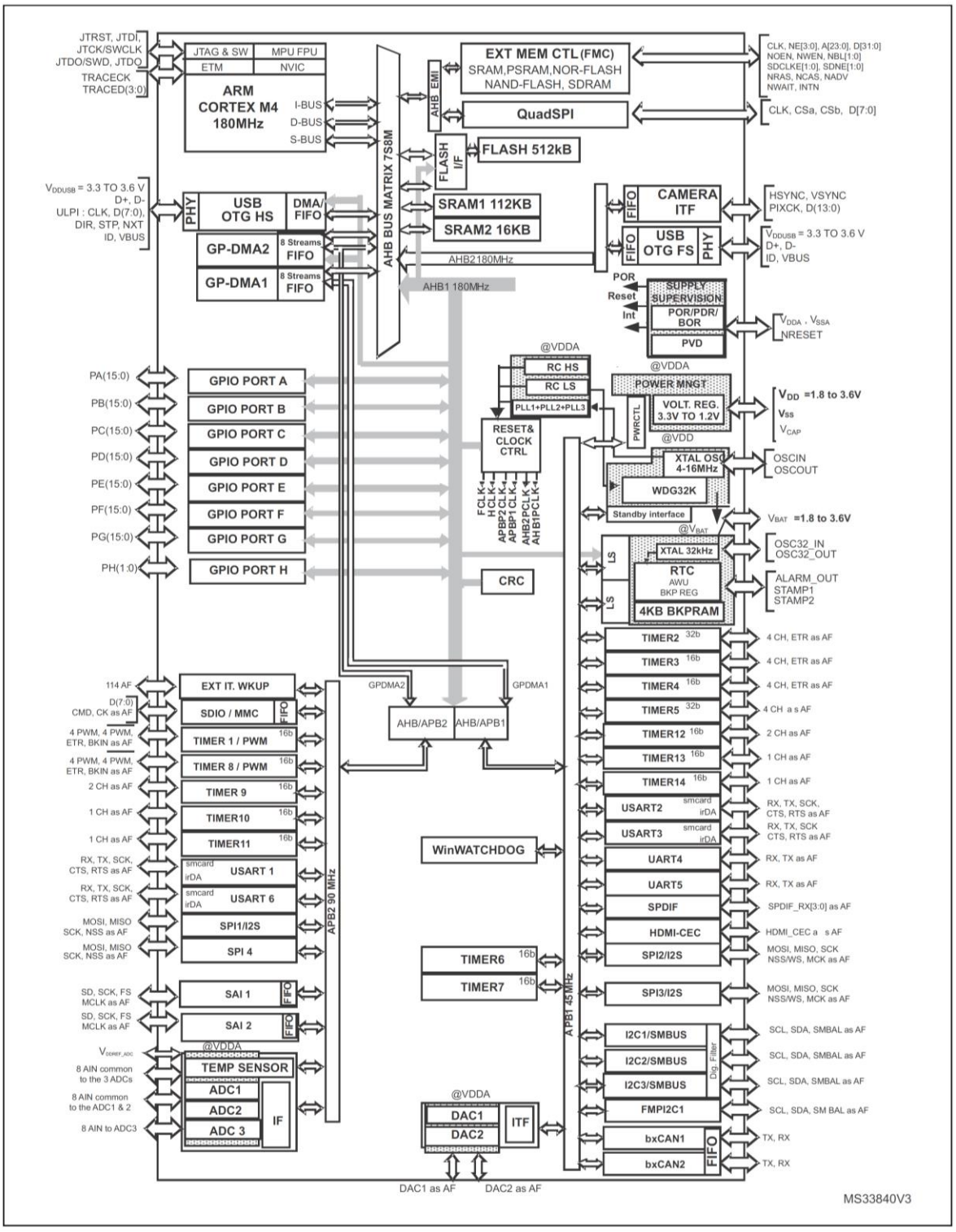


Figure 67: STM32F446 block diagram

Figure 1. STM32L073xx block diagram

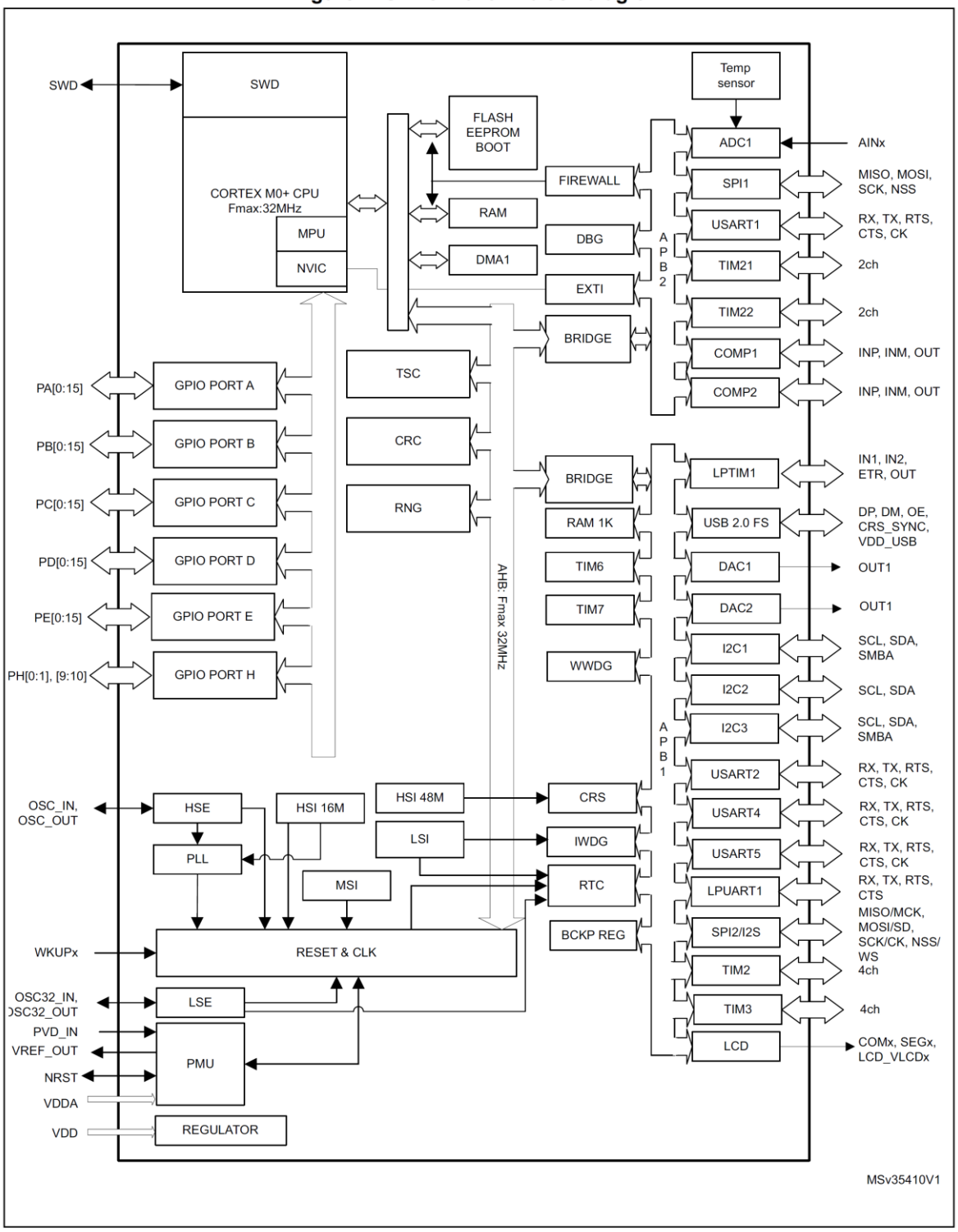


Figure 68: STM32L073 block diagram

Versions

Version 1: June 2021

- Initial release