Application Note -

Using low power modes in STM32F1 microcontrollers

Written by **Marcelo Braga** *and* **João Melga**

Introduction

Embedded systems are designed to perform specific tasks handling limited resources and accomplishing a list of prerequisites, where some of these are directly impacted by the microcontroller power consuming characteristics. Behind various electronic products, there is a microcontroller sourced by limited power supplies, and when it is important to ensure your system will not go down by lack of energy, it is also important to know how you can optimize the power consumption.

This application note was written to give an overview and some example applications of the low-power modes of STM32F1 microcontroller family used to optimize power consumption.

The first part will introduce the different power modes available (Run mode, Sleep mode, Stop mode and Standby mode, to the highest to lowest power consumption) and the correspondent HAL functions related to each mode. After that, an example application will be described since Hardware and Firmware requirements until the code debugging and live testing.

At last, as additional information, you'll learn the basics of the PCC tool available in the STM32CubeMX software, used to estimate power consumption characteristics of projects and an overview of the Tickless Idle Mode for ARM Cortex M3/M4, which enables low-power modes in a multithreading environment with FreeRTOS.

As you will see, each mode has its own properties and trade-offs between power consumption, startup time, available wakeup sources and wakeup time. Is up to the system designers to choose the mode that best suits the application requirements.For more information on this and other features of the STM32F1 microcontroller family, please refer to the STM32F103xx Reference Manual - RM0008, available in the STMicroelectronics website www.st.com.

Contents

List of tables

List of figures

1. Run mode

By default, the microcontroller is set to Run mode after a system or a power Reset. In this mode, the microcontroller is running and has access to all peripherals, making it the most flexible mode, where everything can be enabled and disabled if desired to the trade-off of high power consumption.

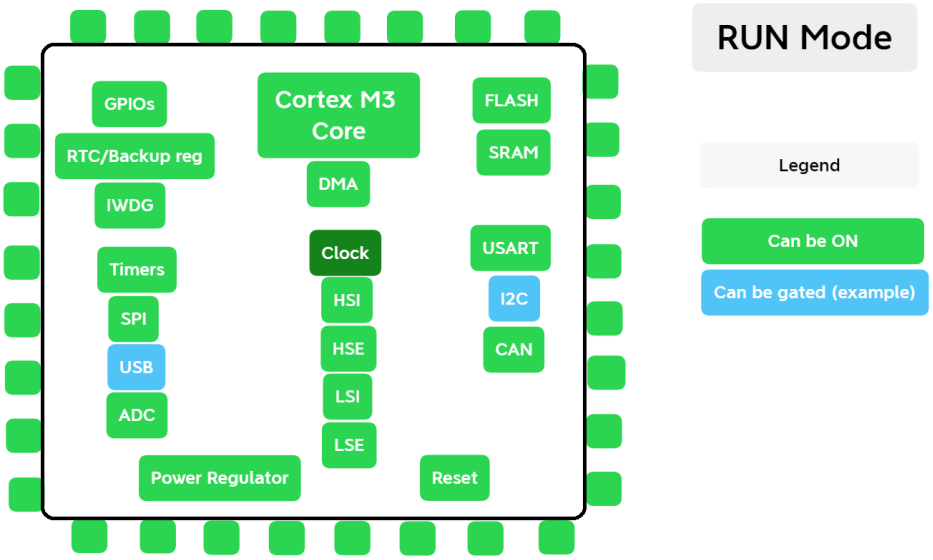

Figure 1: Run Mode

1.1. Reducing power consumption in run mode

Power consumption in Run mode can be reduced by the following methods:

- Slowing down system clocks, because power consumption increases with clock frequency. The speed of the system clocks SYSCLK, HCLK, PCLK1, PCLK2 can be reduced by programming the prescaler registers in the clock tree. Consult the Reference Manual for more information;
- Gating the clocks of APB1, APB2 and AHB peripherals when they are not used, what is done by default;
- Choosing the internal clock source instead of the external ones when possible.

2. Low-power modes

There are three low-power modes available in the STM32F1 family, summarized in the table below.

Mode name	Entry	Wake up	Effect on 1.8V domain clocks	Effect on VDD domain clocks	Voltage regulator
Sleep	WFI / WFE	Any interrupt / Wakeup event	CPU clock OFF no effect on other clocks or analog clock sources	None	ON
Stop	PDDS and LPDS bits + SLEEPDEEP bit + WFI or WFE	Any EXTI line (configured in the EXTI registers)	All 1.8V domain clocks OFF	HSI and HSE oscillators OFF	ON or in low power mode
Standy	PDDS bit + SLEEPDEEP bit + WFI or WFE	WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset	All 1.8V domain clocks OFF	HSI and HSE oscillators OFF	OFF

Table 1: Low power modes

2.1. Sleep mode

In this mode, the core is sent to sleep, turning the CPU clock off. However, other clock sources are not affected, keeping running all peripherals including Cortex-M3 peripherals. All I/O pins keep the same state as in Run Mode. For this reason, this is the low power mode with the highest current consumption compared with the others.

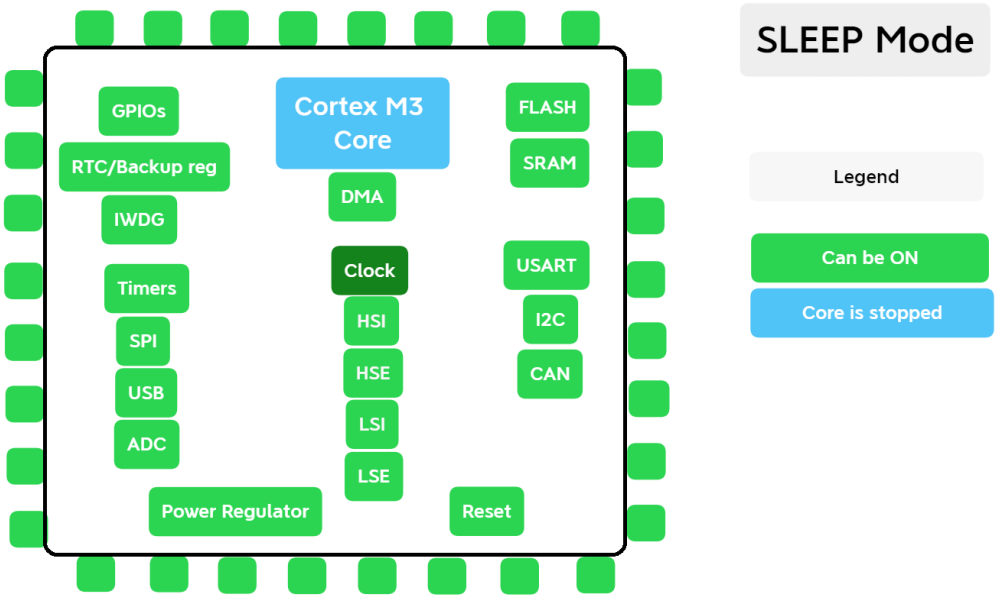

Figure 2: Sleep mode

2.1.1. Sleep Entry / Exit modes

This mode is entered by executing the Wait For Interrupt (WFI) or Wait For Event (WFE) instructions, which set the application wakeup behaviour. There are two entry modes available:

- **Sleep Now**: MCU enters Sleep mode as soon as WFI/WFE instructions are executed. (SLEEPONEXIT bit set to "0").
- **Sleep on Exit**: MCU enters Sleep mode as soon as the lowest priority ISR is executed (SLEEPONEXIT bit set to "1"). This is possible only for the WFI exit mode and makes the system sleep again after wakeup as soon as the woken up interruption has been executed.

And two exit modes:

- **WFI**: any peripheral interrupt acknowledged by the NVIC can wake up the device from the Sleep mode.
- **WFE**: any wakeup event can wake up the device from Sleep mode. Lowest wakeup time since no time is wasted in interrupt entry/exit.
-

2.1.2. Sleep HAL functions

- *HAL_PWR_EnterSLEEPMode*(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFx), where “x” is “I” for the WFI exit mode, or “E” for the WFE exit mode.
- *HAL_PWR_EnableSleepOnExit*() and *HAL_PWR_DisableSleepOnExit*() to change between entry modes.

Sleep now	**Description**
Entry mode	WFI (Wait for Interrupt) or WFE (Wait for Event) while: SLEEPDEEP = 0 and SLEEPONEXIT = 0
Exit mode	WFI -> Interrupt WFE -> Wakeup event
Wakeup latency	None

Table 2: Sleep now

Sleep on exit	**Description**
Entry mode	WFI (Wait for Interrupt) while: SLEEPDEEP = 0 and SLEEPONEXIT = 0
Exit mode	WFI -> Interrupt
Wakeup latency	None

Table 3: Sleep on exit

2.2. Stop mode

In this mode, the CPU is sent to sleep and the HSI and HSE (by consequence, the PLL too) clocks are turned down, leaving available only peripherals with LSI and LSE clocks. However, the SRAM, all registers and GPIO configurations are maintained. A wakeup time due to the clocks reinitialisation is needed.

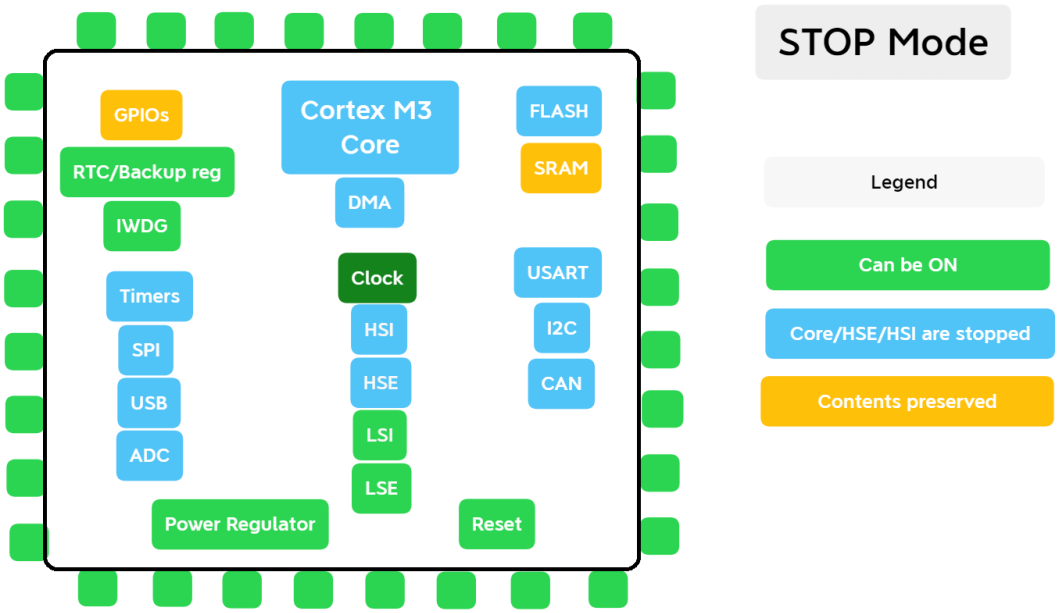

Figure 3: Stop mode

The internal voltage regulator can be set to low power mode to further reduce power consumption by configuring the LPDS bit of the Power control register (PWR_CR). However, the wakeup time will be increased.

To enter Stop mode, all EXTI Line pending bits (in the Pending register EXTI_PR) and RTC Alarm flag must be reset. Otherwise, the Stop mode entry procedure is ignored and program execution continues.

The Stop mode entry is delayed if there's an ongoing memory access or ongoing APB domain access. The following features can be selected in Stop mode by individual control bits:

- IWDG: writing to it's key register or hardware option
- RTC: RTCEN bit in RCC_BDCR
- LSI oscillator: LSION bit in RCC_CSR
- LSE oscillator: LSEON bit in RCC_BDCR

2.2.1. Stop exit modes

The same modes of Sleep mode are available: Wait For Interrupt (WFI) and Wait For Event (WFE).

2.2.2. Stop HAL functions

- HAL_PWR_EnterSTOPMode(PWR_REGULATOR_VALUE, PWR_SLEEPENTRY_WFx), where “x” is “I” for the WFI exit mode, or “E” for the WFE exit mode and PWR_REGULATOR_VALUE could be PWR_MAINREGULATOR_ON or PWR_LOWPOWERREGULATOR_ON

Stop mode	**Description**
Entry mode	WFI or WFE while: – SLEEPDEEP = 1 – PDDS = 0 in PWR_CR – Select the voltage regulator mode by configuring LPDS bit in PWR_CR
Exit mode	WFI -> Any EXTI Line configured in Interrupt mode WFE -> Any EXTI Line configured in event mode
Wakeup latency	HSI RC wakeup time + Regulator wakeup time from low-power mode

Table 4: Stop Mode

2.3. Standby mode

Also known as DeepSleep mode, in the Standby mode the voltage regulator is turned off, shutting down the 1.8V power domain and consequently, the CPU, the HSI and HSE oscillators and all peripherals. Only the RTC, IWDG, Reset and WakeUp (WKUP) buttons are enabled. The SRAM, registers (except Backup registers) and GPIO configurations are lost unless a dedicated standby circuitry is implemented

The wakeup latency is the same as a system Reset latency, since all configurations need to be executed after leaving the Standby mode (after waking up, program execution restarts in the same way as a Reset).

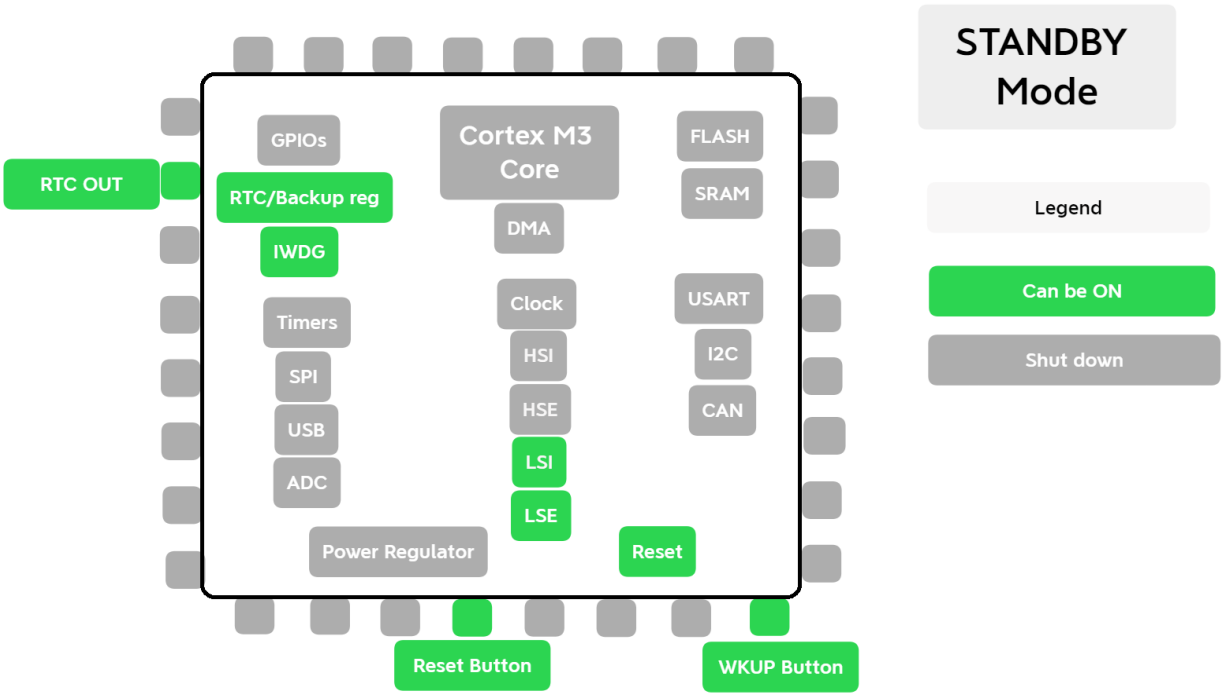

Figure 4: Standby mode

The following features can be selected in Standby mode by individual control bits:

- IWDG: writing to it's key register or hardware option
- RTC: RTCEN bit in RCC_BDCR
- LSI oscillator: LSION bit in RCC_CSR
- LSE oscillator: LSEON bit in RCC_BDCR

2.3.1. Standby exit modes

There are four ways to wake up/exit the Standby mode:

- Reset button pressed (NRST pin)
- IWDG Reset, if previously defined
- WKUP button pressed (rising edge in WKUP pin)
- RTC alarm occurs, if previously defined

Standby mode	Description
Entry mode	WFI or WFE while: – SLEEPDEEP = 1 – PDDS = 1 in PWR_CR – WUF = 0 in PWR_CSR
Exit mode	WKUP pin rising edge, RTC alarm, external Reset in NRST pin, IWDG Reset.
Wakeup latency	Regulator start up + Reset phase

Table 5: Standby mode

2.3.2. HAL functions

- HAL_PWR_EnterSTANDBYMODE();
- HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1), each microcontroller has one or more wakeup pins and you can find it in the respective Reference Manual;
- HA_PWR_Disable_WakeUpPin(PWR_WAKEUP_PIN1).

2.4. Configuring Auto-wakeup from Low-power modes

As shown above, the RTC can be used in all modes to generate an alarm, in a programmable time base, and wake up from the low-power modes. For that, you must:

a. Configure the EXTI [17] to be sensitive to rising edge;
b. Configure the RTC to generate an alarm, using the HAL_RTC_SetAlarm_IT() function.

Two of the three RTC clock sources can be selected by programming the RTCSEL bits in RCC_BDCR or the HAL_RCC_OscConfig():

- LSE : precise time base 32.768 kHz external crystal oscillator, with less than 1uA added consumption;
- LSI : ~40kHz internal RC oscillator, designed to add minimum power consumption.

3. Low power application example

To better understand how and when to use each one of the three low power modes available in the STM32F1 microcontrollers, a demo application has been provided with this Application Note. All files and libraries quoted in the following paragraphs could be accessed in the **GitHub repository** described in *References* topic.

Basically, the application main() function consists in a finite state machine with three possible states: SLEEP_TESTING_STATE, STOP_TESTING_STATE and STANDBY_TESTING_STATE. Each one is responsible to set correctly the respective low power mode and enable you to test it. The current state and the interruptions handling are sinalized by LEDs blinking or being toggled and every wakeup is triggered by buttons being pressed, as described in the topic *Testing application*. Beyond the LEDs, the application behaviour could be explored by debugging it at all.

3.1. Hardware and Firmware requirements and setup

The demo has been developed under the stack STM32CubeMX + SystemWorkbench for STM32 IDE (SW4STM32) + HAL api functions and tested in a STM32F103C8T6 microcontroller available in a common BluePill board. You can download SW4STM32 IDE and STM32CubeMX in the following links:

- https://www.st.com/en/development-tools/sw4stm32.html;
- https://www.st.com/en/development-tools/stm32cubemx.html.

If you open the file *STM32F103xx-LowPowerModes.ioc* In STM32CubeMX, you'll be able to see all pinout, clocks and other resources configurations. In order to assemble circuit parts properly and test demo application, you should follow STM32CubeMX pinout diagram - shown as soon as you open the project - and the circuit schematics below, which requires the following materials.

Material	Quantity	Usability
Push buttons	3	Trigger wakeups
LEDs	2	Indicate states and interruptions being handling
Resistors	2	Values between 200 and 2,2K Ohms may be sufficient to downgrade the LEDs current
BluePill board controlled by a STM32F103C8T6 MCU	1	Run the application at all
ST-Link V2 debugger and programmer	1	Debug and upload programs to the BluePill board
Male-male jumpers	5+	Connect BluePill to buttons and LEDs
Female-female jumpers	4	Connect ST-Link V2 to BluePill
Protoboard	1	The base were we'll assemble the circuit

Table 6: Demo application required materials

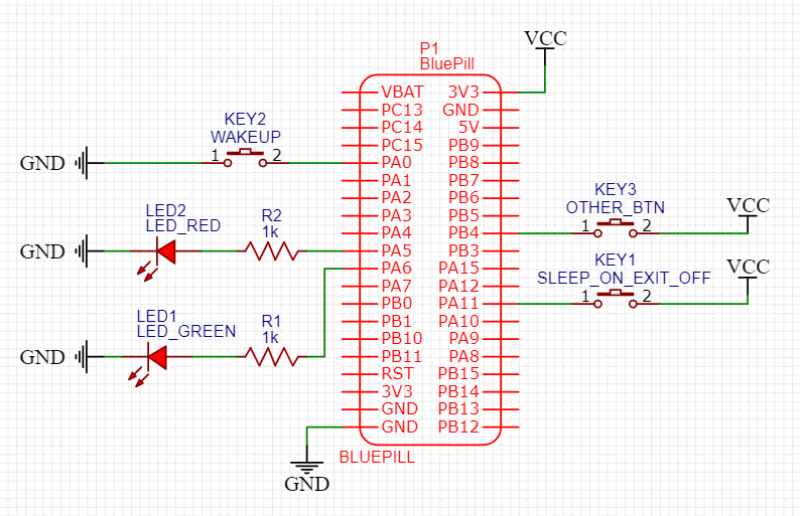

Figure 5: Demo circuit schematics

3.2. Configuring STM32CubeMX

As you can see in the topics above, we'll need to set 5 pins to test the demo - 2 outputs and 3 inputs, what has been already done in *STM32F103xx-LowPowerModes.ioc,* but if you want to reproduce the steps to create this file by yourself in STM32CubeMX, you must follow the steps below:

a. Create a new CubeMX project going to *File -> New Project -> MCU/MPU Selector* and type "*STM32F103C8*", double click the result item in MCU/MPU list;

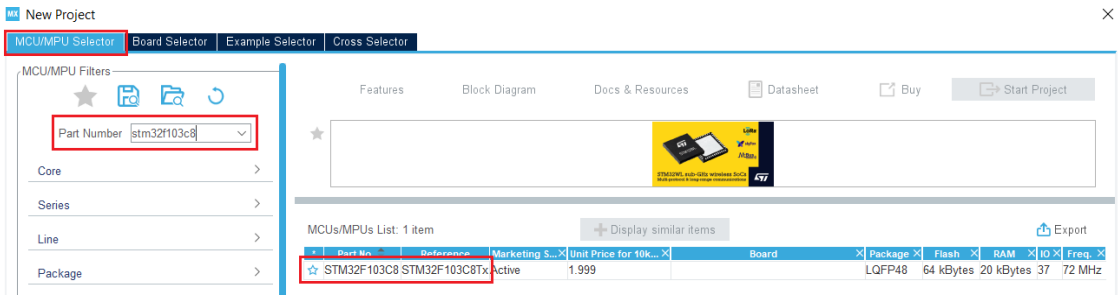

Figure 6: STM32CubeMX, creating new project

b. To configure the main five GPIOs used in demo, go to Pinout & Configuration menu and repeat this process:
   - Click once on the pin with the mouse's left button and **select a State** (PA5 and PA6 - *GPIO_Output*, PA0 - *SYS_WKUP*, PA11 - *GPIO_EXTI11* and PB4 - *GPIO_EXTI4*);
   - Click again on the pin, but now with the right button, then select ***Enter User Label*** *(*PA5 - *LED_RED*, PA6 - *LED_GREEN*, PA0 - WAKEUP, PA11 - *SLEEP_ON_EXIT_OFF* and PB4 - *OTHER_BTN*);
   - Go to *System Core -> GPIO -> tab GPIO*, click in a row of the table and configure the property **GPIO Pull up/Pull down** (PA11 to *Pull-down* and PB4 to *Pull-down, only*). Now you must see something like follows (WAKEUP pin is shown in *System Core -> GPIO -> tab SYS*);

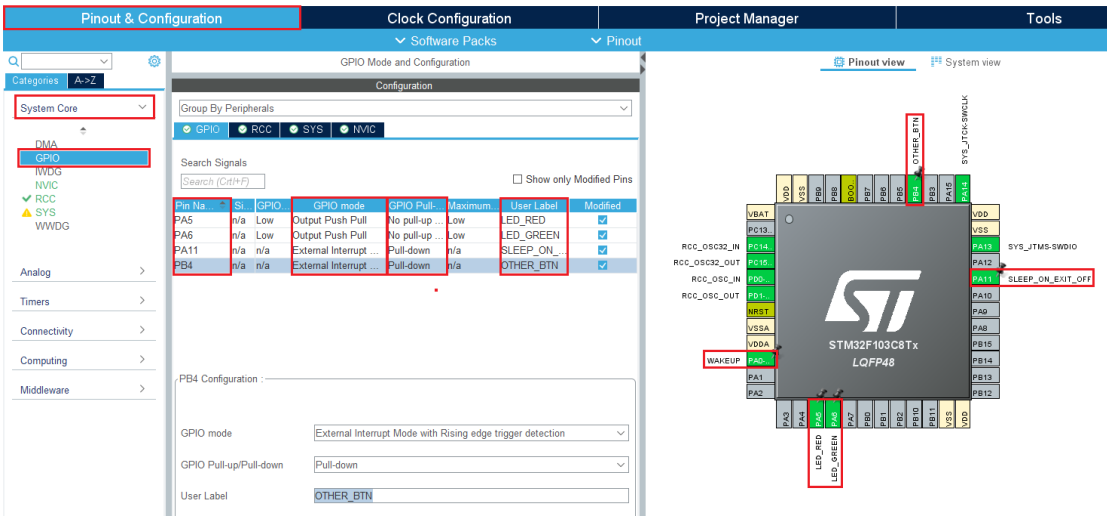

Figure 7: STM32CubeMX, configuring pins

- And now we can't forget to activate the **EXTI interruptions** (used in SLEEP_ON_EXIT_OFF and OTHER_BTN). Go to *System Core* -> *NVIC* and enable *EXTI line4 interrupt* and *EXTI line[15:10] interrupts*, like below;

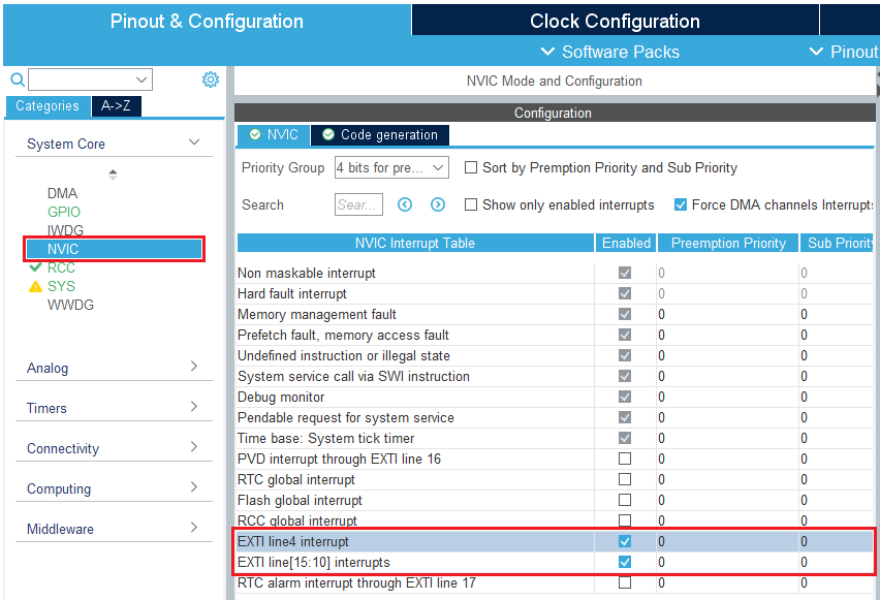

Figure 8: STM32CubeMX, activating NVIC interruptions

c. Now let's activate the system debug tools going to *System Core* -> *SYS* -> Mode, set ***Debug*** to *Serial Wire and* ***Timebase Source*** to *SysTick*;

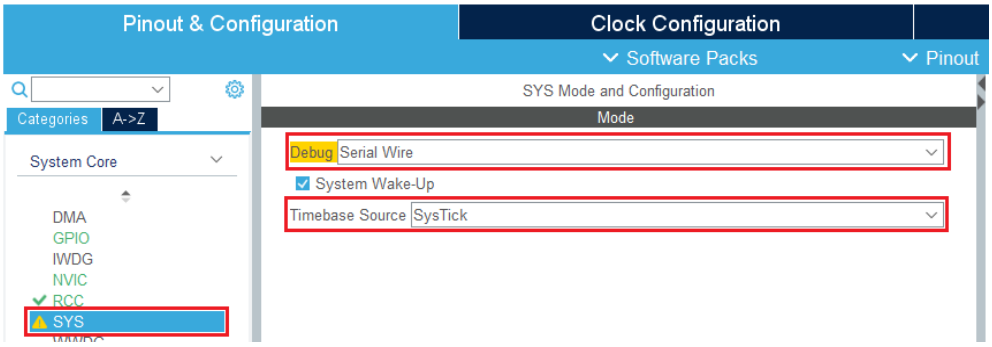

Figure 9: STM32CubeMX, activating debug tools

d. For this specific demo application we won't need to change *Clock Configurations*, so the next step is to tell STM32CubeMX how we would like to generate our project files. As we are developing under SW4STM32 IDE, go to *Project Manager -> Project Settings* and set ***Toolchain / IDE*** to SW4STM32 and mark the ***Generate Under Root*** checkbox;

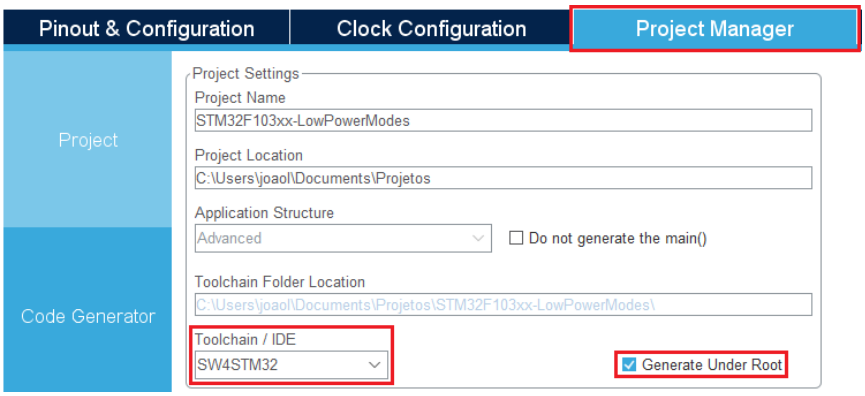

Figure 10: STM32CubeMX, configuring project settings

e. The last step is to generate our c / c++ project files. We just need to hit the big and blue ***GENERATE CODE*** button available in the top menu and STM32CubeMX does all the hard work for us. Before that, we'll concentrate on developing and testing the application into SW4STM32.

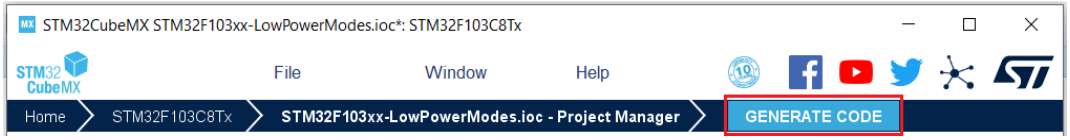

Figure 11: STM32CubeMX, generating code

3.3. Software description

The demo application software is divided in three separated parts that access the same global variable *state*.

a. A loop with a finite state machine activating each of the low power modes, performing changes in *state* and blinking LEDs, as shown in the code below. Notice how *Low power wake up routine* and *States changes and indications* of *STANDBY* mode are not in the main loop. This is happening because STANDBY woken up process makes the code run from the first line, therefore the segment described in the next topic fulfills these roles.

```
/* USER CODE BEGIN WHILE */
while (1) {
  HAL_GPIO_WritePin(GPIOA, LED_GREEN_Pin, GPIO_PIN_RESET);
  HAL_GPIO_WritePin(GPIOA, LED_RED_Pin, GPIO_PIN_RESET);

  switch (state) {
  case SLEEP_TESTING_STATE:
    // LED_GREEN will blink 3 times = System is about to enter in SLEEP mode and SleepOnExit is enabled
    GPIO_Blink(GPIOA, LED_GREEN_Pin, 3, 500);

    HAL_SuspendTick();
    HAL_PWR_EnableSleepOnExit();

    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);

    HAL_ResumeTick();

    // LED_GREEN will blink 15 times quickly = System has just woken up from SLEEP mode and SleepOnExit is disabled
    GPIO_Blink(GPIOA, LED_GREEN_Pin, 15, 100);
    state = STOP_TESTING_STATE;
    break;

  case STOP_TESTING_STATE:
    // LED_RED will blink 3 times = System is about to enter in STOP mode and SleepOnExit is enabled
    GPIO_Blink(GPIOA, LED_RED_Pin, 3, 500);

    HAL_SuspendTick();
    HAL_PWR_EnableSleepOnExit();

    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);

    SystemClock_Config(); // SystemClock may be configured again, because it was disabled
    HAL_ResumeTick();

    // LED_RED will blink 15 times quickly = System has just woken up from STOP mode and SleepOnExit is disabled
    GPIO_Blink(GPIOA, LED_RED_Pin, 15, 100);
    state = STANDBY_TESTING_STATE;
    break;

  case STANDBY_TESTING_STATE:
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);

    HAL_PWR_EnterSTANDBYMode();
  }
```

Labels:
Preparing to low power routine
Low power activation routine
Low power wakeup routine
States changes and indications

Figure 12: Demo main finite state machine routine

b. A routine to handle STANDBY wakeups - clearing *PWR_FLAG_SB* and disabling *PWR_WAKEUP_PIN1* (A0) - and indicate whenever the system has been reseted or just woken up from a *deep sleep*;

```
/* USER CODE BEGIN 2 */
if (__HAL_PWR_GET_FLAG(PWR_FLAG_SB) != RESET) {
  // If PWR FLAG SB was not RESET, system has been waked up in STANDBY mode
  __HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);
  HAL_PWR_DisableWakeUpPin(PWR_WAKEUP_PIN1);

  // LED_RED will blink once very slowly = System has been waked up from STANDBY mode
  //                                       and is about to deep sleep again if SLEEP_ON_EXIT_OFF
  //                                       button is not pressed
  GPIO_Blink(GPIOA, LED_RED_Pin, 1, 2000);
  state = STANDBY_TESTING_STATE;
}

else {
  // LED_GREEN will blink once very slowly = System has been powered on or reseted
  GPIO_Blink(GPIOA, LED_GREEN_Pin, 1, 2000);
  state = SLEEP_TESTING_STATE;
}

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
```

Labels:
Preparing to low power routine
Low power activation routine
Low power wakeup routine
States changes and indications

Figure 13: Demo main standby routine a

c. The buttons' ISR, which has different behaviour for clicks in SLEEP_ON_EXIT_OFF or OTHER_BTN buttons. In general, states are changed just if the SLEEP_ON_EXIT_OFF button is pressed, either by calling *HAL_PWR_DisableSleepOnExit() - as in* SLEEP and STOP modes - or just setting another value to *state,* like in *STANDBY* mode.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  // In SLEEP and STOP testing states, turning off SleepOnExit will make main() code run again
  // Otherwise, the interruption would be executed and the MCU would sleep without look to main()
  switch (state) {
  case SLEEP_TESTING_STATE:
    if (GPIO_Pin == SLEEP_ON_EXIT_OFF_Pin)
      HAL_PWR_DisableSleepOnExit();
    else
      // LED_GREEN will be toggled = System has just woken up from SLEEP mode and SleepOnExit stills enabled
      HAL_GPIO_TogglePin(GPIOA, LED_GREEN_Pin);

    break;

  case STOP_TESTING_STATE:
    if (GPIO_Pin == SLEEP_ON_EXIT_OFF_Pin)
      HAL_PWR_DisableSleepOnExit();
    else
      // LED_RED will be toggled = System has just woken up from STOP mode and SleepOnExit stills enabled
      HAL_GPIO_TogglePin(GPIOA, LED_RED_Pin);

    break;

  case STANDBY_TESTING_STATE:
    if (GPIO_Pin == SLEEP_ON_EXIT_OFF_Pin)
      state = SLEEP_TESTING_STATE; // Changing the state is the way we avoid STANDBY to be performed again

    break;
  }
}
```

Labels:
Preparing to low power routine
Low power activation routine
Low power wakeup routine
States changes and indications

Figure 14: Demo main standby routine b

3.4. Testing application

In order to test the application using Wire Debug, in SW4STM32 go to *Run -> Debug Configurations -> Ac6 STM32 Debugging -> STM32F103xx-LowPowerModes -> Debugger -> Show Generation Options* and set ***Reset Mode*** to *Software System Reset*, like below.

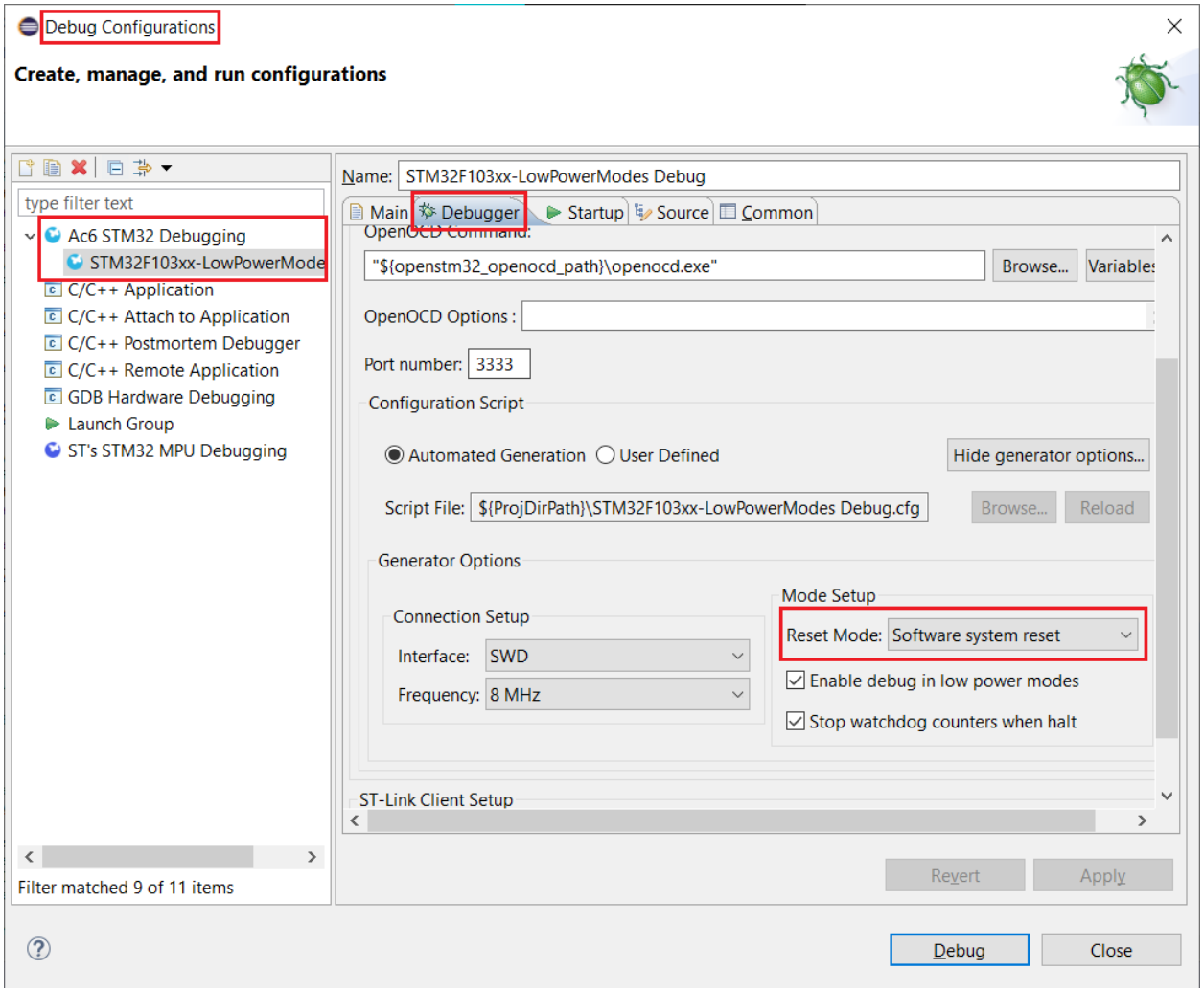

Figure 15: Demo debug configuration

After that, connect the STLink-V2 to the computer and the BluePill board and click on ***Debug***. Your board must behave like described in the next table.

What's happening?	Indicator	Trigger
Powered on or hard reseted	LED_GREEN blink once very slowly	Power on
About to enter in SLEEP and SleepOnExit is enabled	LED_GREEN blink 3 times	Power on or SLEEP_ON_EXIT_OFF interruption after STANDBY woken up
Waked up from SLEEP and SleepOnExit stills enabled	LED_GREEN toggled	OTHER_BTN interruption
Waked up from SLEEP and SleepOnExit is disabled	LED_GREEN blink 15 times quickly	SLEEP_ON_EXIT_OFF interruption
About to enter in STOP and SleepOnExit is enabled	LED_RED blink 3 times	Waked up from SLEEP and SleepOnExit is disabled
Waked up from STOP and SleepOnExit stills enabled	LED_RED toggled	OTHER_BTN interruption
Waked up from STOP and SleepOnExit is disabled	LED_RED blink 15 times quickly	SLEEP_ON_EXIT_OFF interruption
Waked up from STANDBY and is about to deep sleep again if SLEEP_ON_EXIT_OFF not pressed	LED_RED blink once very slowly	Waked up from STANDBY or Waked up STOP mode and SleepOnExit is disabled

Table 7: Demo application states, indicators and triggers

In summary, the first time you start the application, the LED_GREEN will turn on and turn off after some seconds, indicating the system is not waking up from a deep sleep / standby. Right after that, the LED_GREEN will blink 3 times, showing us the application is about to enter SLEEP mode (because we chose SLEEP_TESTING_STATE as the first state) with SleepOnExit enabled.

From this point on, we have two options: click on SLEEP_ON_EXIT_OFF or OTHER_BTN. If you click this last, LED_GREEN will just be toggled, signaling the EXTI ISR has been executed, but not the code inside main() function. However, if SLEEP_ON_EXIT_OFF has been clicked, LED_GREEN will blink quickly 15 times, which means the system didn't sleep right after executing the ISR and the main() function code has been executed.

When the main() function is executed and state is SLEEP_TESTING_STATE, it becomes STOP_TESTING_STATE and LED_RED will blink 3 times, showing us the application is about to enter STOP mode with SleepOnExit enabled. Just link in SLEEP mode, if we click OTHER_BTN, LED_RED is toggled and main() routine is not executed, but if push SLEEP_ON_EXIT_OFF button, LED_RED blink quickly 15 times, main() is executed and state becomes STANDBY_TESTING_STATE.

In the STANDBY_TESTING_STATE the application behaviour is a little bit different, because the woken up interruptions in the STANDBY mode are also a bit more limited. For this reason, the system wakes up just if the WAKEUP button is pressed. Otherwise, nothing will happen. If after the system wakes up from STANDBY the SLEEP_ON_EXIT_OFF button is pressed, state becomes SLEEP_TESTING_STATE and all the logic described above repeats itself.

The GitHub repository quoted before has some animations and additional descriptions that may help you understand even better the low power modes.

4. Estimating power consumption with PCC tool in CubeMX

Estimating power consumption and battery life in microcontrolled projects can be difficult due to the variety of resources draining power from battery. The PCC (Power Consumption Calculator) tool, available in the CubeMX software is a viable option to manage power consumption in the designing phases of your project.

After selecting your MCU/Board, setting all the necessary configurations for your project in the “Pinout & Configuration” and “Clock Configurations” tabs, you can jump straight to the “Tools” tab, where PCC is allocated. The PCC tool will consider all the set up done in the previous configuration tabs by default, however, you can modify them freely if wanted.

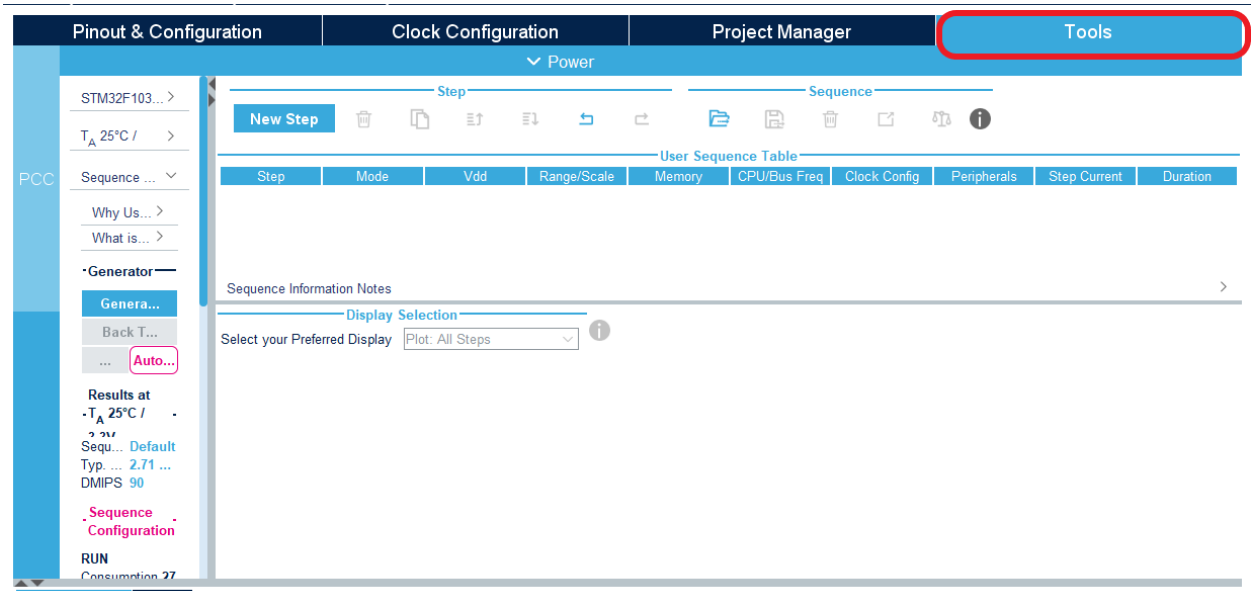

Figure 16: PCC tool in CubeMX

The PCC tool has a variety of functionalities, being the basic ones:

- Select the VDD value of the MCU
- Select a battery from the available options or specify your own
- Create steps in the current consumption X time graph, representing the consumption of your project.

- After all steps are created, the tool will indicate Sequence Time / Ta Max, Average current Consumption, Battery Life Estimation and Average DMIPS.

4.1 Battery Life Estimation Example

Let's estimate the Battery Life of a fully charged A3400 battery (3400 mAh) on a simple application on a STM32F103C8T6 MCU, which for each second, does an analog to digital conversion at 12Mhz and send the data via I2C to another application.

Since these operations are fast, there's no need to keep the MCU at RUN mode all the time. So, for demonstration purposes only, let's consider the duration of this operation 10ms, and MCU clock the maximum value of 72MHz during RUN mode. The rest of the time, the CPU will be at STOP mode. Every second, the application changes from STOP to RUN mode, executes the task in 10ms, and comes back to STOP mode.

In the STM32CubeMX we selected the STM32F103C8T6 MCU, enabled the High Speed Clock (HSE) with the Crystal/Ceramic Resonator, configured CPU Clock at 72MHz and ADC1 to 12MHz. The RTC, all ADC1 channels and I2C1 were enabled.

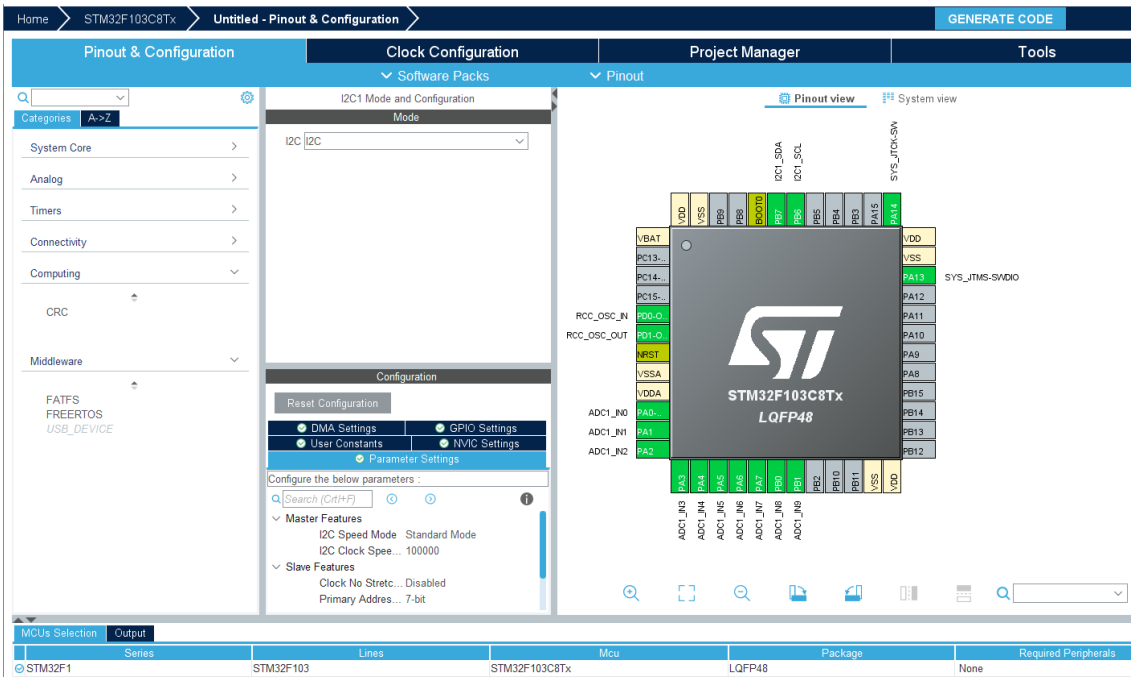

Figure 17: PCC tool example - CubeMX configuration

In the PCC tool, all power consumed by the features mentioned above are already considered, only requiring us to establish the RUN to STOP modes ratio (in our example, 0.01:1). If you want, the PCC allows the user to add any additional amount of power consumption if needed. Let's create a default sequence, and change the steps ratio accordingly to the following figure.

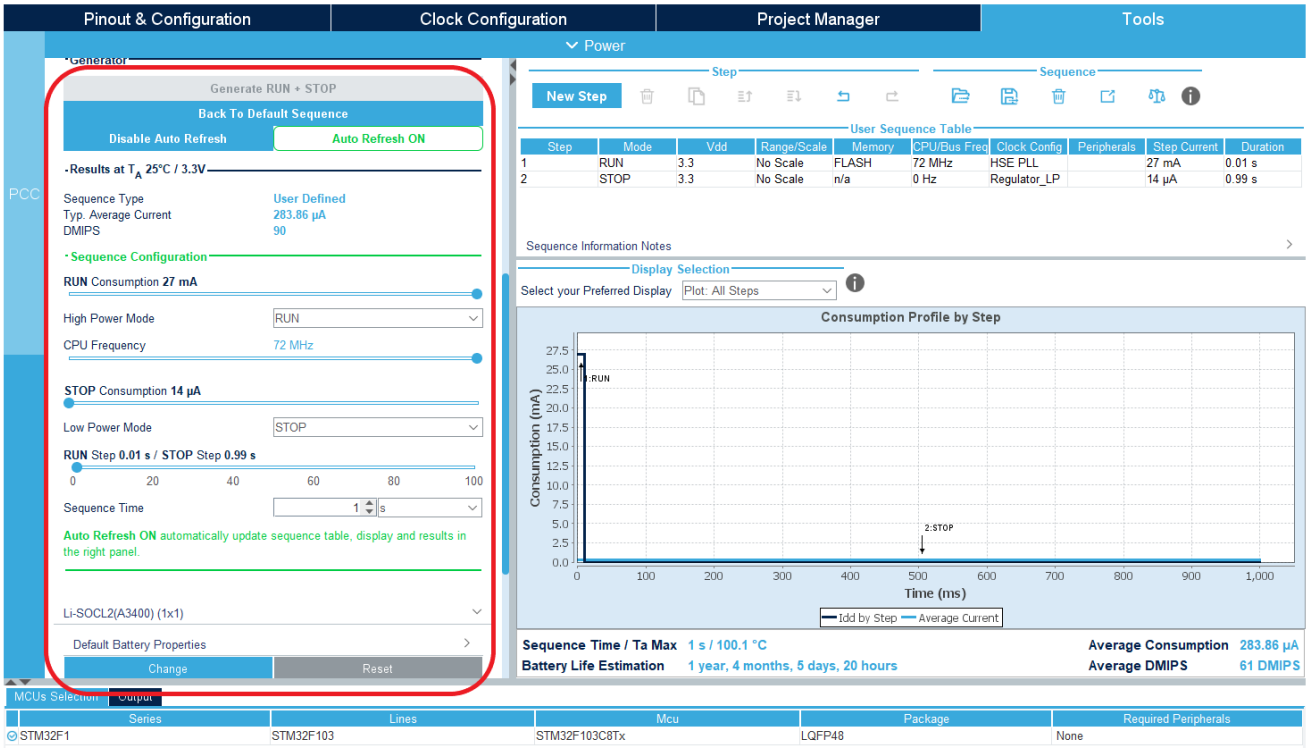

Figure 18: PCC tool example - PCC configuration

After all these steps, we can see in the right the Consumption Profile graph of the project and, in the graph's bottom, the generated estimations. In our example, the battery would last roughly **1 year and 4 months**, with an average current consumption of 283,86 uA. If RUN mode was on 100% of the time, the battery wouldn't last a week, consuming 27mA for a total of **5 days and 5 hours**. That shows, although superficially, the importance of low-power modes in limited current source applications.

5. Low-power modes with FreeRTOS

Saving power in multithreading applications can be difficult to manage, since real time systems need periodic timing interrupts, default every millisecond, to change context between tasks. However, with FreeRTOS, it's possible to enable the **Tickless Idle Mode to save battery during periods in which no tasks are on Run state**.

In FreeRTOS, the Idle task is created automatically by the RTOS scheduler to ensure at least one task is running. This task is created with the lowest priority to ensure the CPU will only be Idle when no other tasks are available. We can use the Idle task to put the CPU to a low-power mode, however, at every RTOS tick, the CPU would exit the low-power mode, making it change between Run and Low-power modes at each RTOS tick until a new task is ready. So, in the default configuration, FreeRTOS is not very battery friendly.

To ensure the low-power mode in Idle stays for a longer period of time, the RTOS tick needs to be stopped. However, if not turned on again, the RTOS scheduler would not work, freezing the whole application. How to manage this situation?

The Tickless Idle Mode in FreeRTOS resolves this problem, implementing first, the **portSUPRESS_TICKS_AND_SLEEP()** function, that suspend the ticks and puts the MCU in Sleep WFI mode (the user can modify this function and configure any other low-power mode. ARM Cortex M3/M4 have a default function already). The portSUPRESS_TICKS_AND_SLEEP() function has a **xExpectedIdleTime** parameter, which indicates an estimation, done by the RTOS, of how much time the application will stay in the Idle task, and therefore, in the Sleep mode.

In that way, when there's no other tasks in Ready state, the system would enter the Idle task and Sleep mode for xExpectedIdleTime ticks. After this time has elapsed, the tick interrupt is restarted and the tick count value is adjusted. Note that this is an automatic wakeup mechanism, but nothing stops the designer from implementing other ways of waking up the system such as any interrupt with WFI, not needing the xExpectedIdleTime.

We can enable the Tickless Idle Mode, in its default state, defining configUSE_TICKLESS_IDLE as 1 in FreeRTOSConfig.h file. We can do this in the CubeMX software also, as shown below.

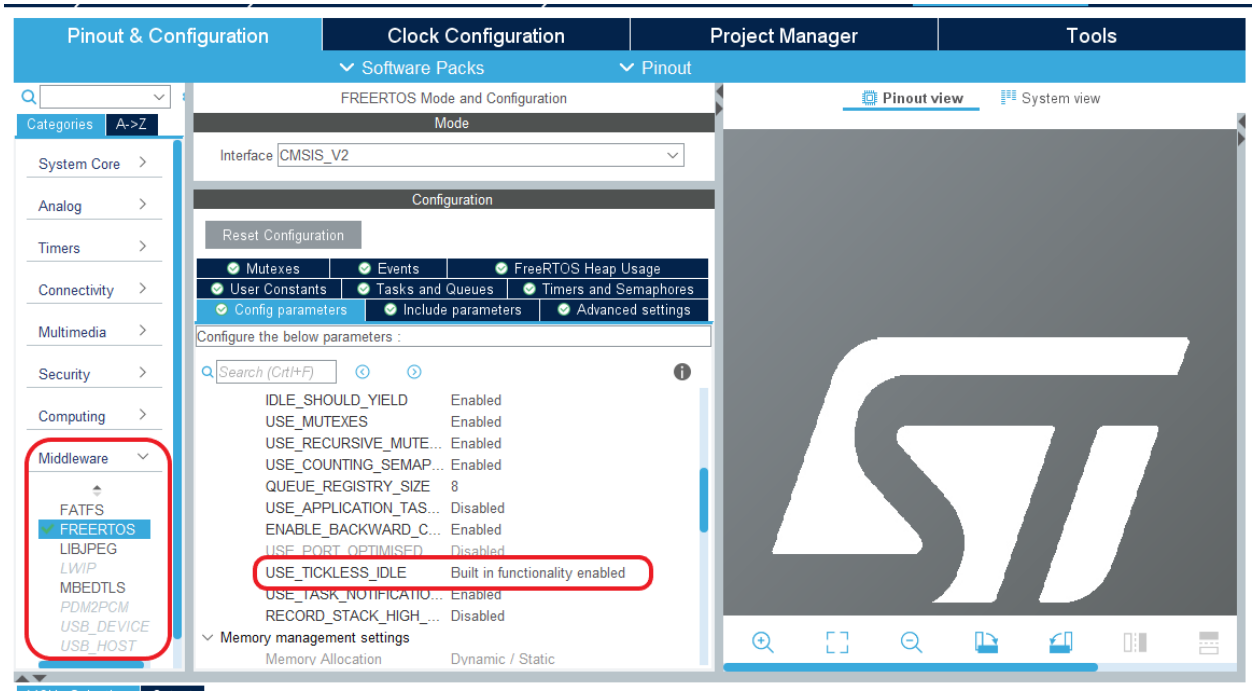

Figure 19: Tickless Idle Mode in CubeMX

The CubeMX also creates two Macros, one that executes immediately before entering the Idle Tickless task, and another immediately after, that can be changed by the user. These can be useful to set up the low-power mode as wished before entering the Idle, disabling power hungry peripherals, slowing down peripherals clocks, enabling any necessary wakeup features, and reversing these changes after exiting. If the auto-wake up mechanism is not needed, set the xExpectedIdleTIme variable to 0 in the PreSleepProcesing() function and define your own wake up mechanism.

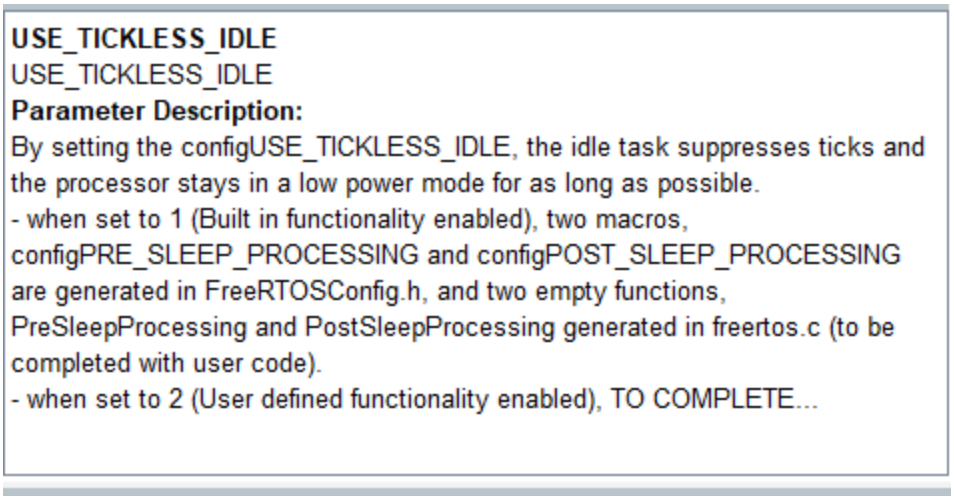

Figure 20: Tickless Idle Mode description in CubeMX

For any other information on this topic, please refer to *References* and FreeRTOS website on: freertos.org.

6. Conclusion

The low-power modes available in the STM32F1 family are a powerful tool in systems design, making it possible to accomplish a key requirement in embedded systems: battery efficiency. Although powerful,, a developer needs to have great knowledge and mastery in order to properly use the Sleep, Stop and Standby modes, since not every MCU feature is available. It must be very clear which MCU features are needed during the modes, which wakeup features are available and which wakeup time is acceptable in order to choose the right mode that fits the application requirements.

7. References

- Low power modes demo (CubeMX + SW4STM32 + HAL api) GitHub repository. Available on: https://github.com/joaomelga/STM32F103xx-LowPowerModes
- Application Note 4777 - STM32 power modes examples. Available on: https://www.st.com/resource/en/application_note/dm00237631-stm32-power-mode-examples-stmicroelectronics.pdf
- Application Note 2629 - STM32F101xx, STM32F102xx and STM32F103xx low-power modes. Available on: https://www.st.com/resource/en/application_note/cd00171691-stm32f101xx-stm32f102xx-and-stm32f103xx-lowpower-modes-stmicroelectronics.pdf
- STM32F103xx Reference Manual - RM0008. Available on: www.st.com
- Low power modes available on STM32. Available on: https://controllerstech.com/low-power-modes-in-stm32.
- Low Power Support - Tickless Idle Task. Available on: https://www.freertos.org/low-power-tickless-rtos.html#:~:text=The%20FreeRTOS%20tickless%20idle%20mode,the%20tick%20interrupt%20is%20restarted.
- Low Power RTOS For ARM Cortex-M MCU's. Available on: https://www.freertos.org/low-power-ARM-cortex-rtos.html

7. Revision history

Date	Revision	Changes
08-Mar-2021	1	Initial Release

Table 8: Revision history