Laboratory 2: Communicating with PC and on-board components
System-On-Chip Architecture

Walter Gallego[1] Andrea Pignata[2] Gianvito Urgese[1]

[1]name.lastname@polito.it
[2]name_lastname@polito.it

Abstract

In this laboratory session, we will explore the use of some components in the stm32f4discovery board. In particular, the serial communication using the usb-otg connection, the audio codec, the accelerometer sensor, the user leds and the user and reset buttons.

We will also explore more advanced management of interrupts, and how they relate to the low power modes of a typical microcontroller.

Keywords: *Serial communication, audio codec, accelerometer, GPIO, CubeMX, Platformio, Interrupts, Low Power Modes*

Contents

1 Guided section

1.1 Materials

This guide is accompanied by a CubeMx / platformio project that will be used as starting point for the laboratory.

Students are encouraged to use the provided project, but they can also start from scratch if they prefer.

1.2 Project description

The block diagram of the system is shown in Figure 1. It consists of one board connected via serial port to a Host PC, that can send commands and receive feedback from the board. The system will use the following components of the board:

- The 4 user LEDs in the board
- The micro usb connection, for usb-otg configured as Virtual Com Port, for serial connection with the Host PC.
- The audio codec, to generate audible signals (sine functions of proper frequency) that are delivered to the audio jack in the board.
- The accelerometer sensor.
- The user and reset buttons.

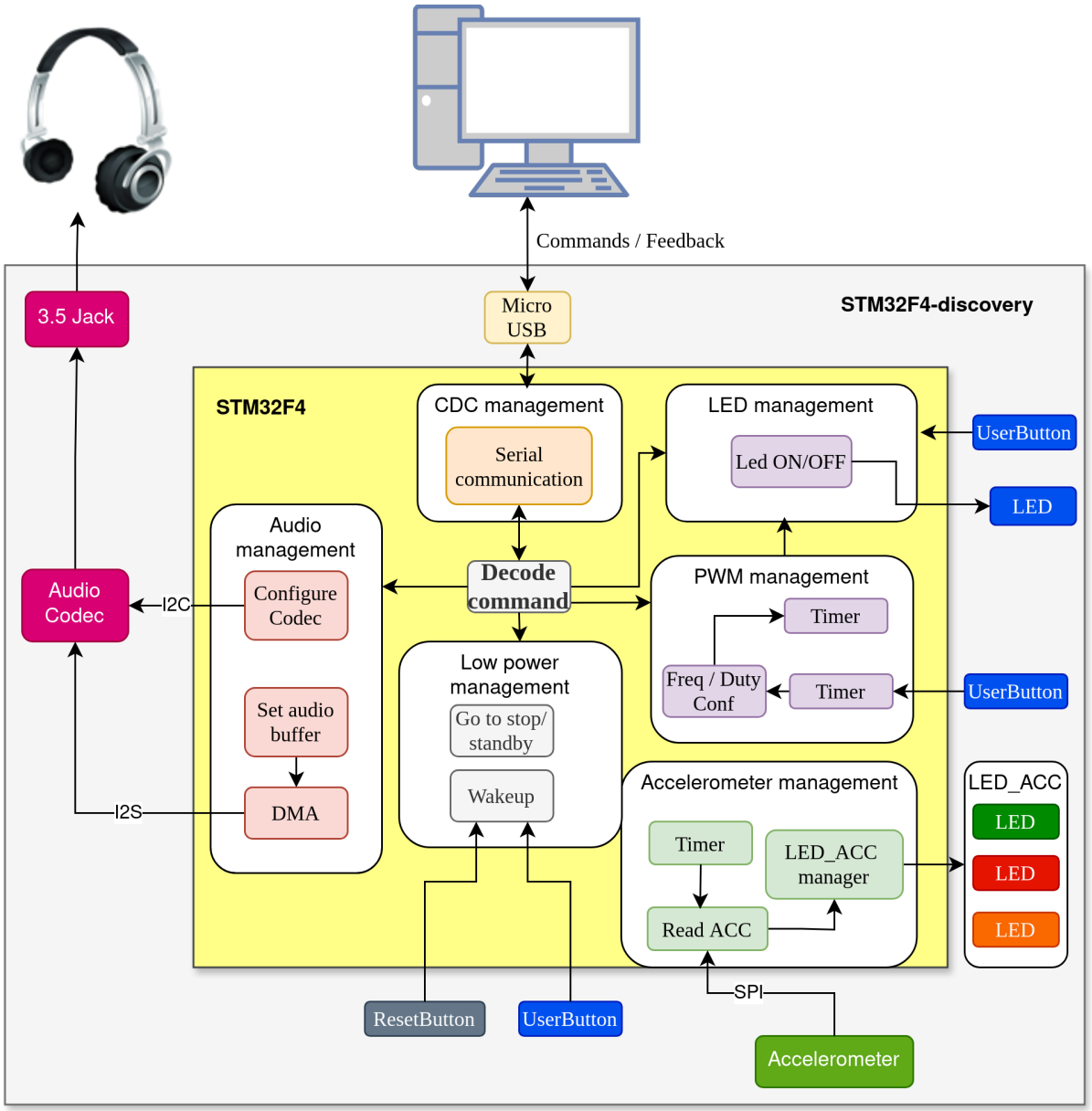

Figure 1: Block diagram of the system

1.2.1 Commands via Serial connection

- From the computer, the user can send commands to the Board.
- The commands sent from the PC can have any length. The board recognizes a command when the enter key is pressed on the host.
- If the command is not correct, the board will send an error message to the PC. If the command is correct, it will execute it.
- Possible commands are listed in table 1.

1.2.2 PWM generation

- The Board generates a PWM signal, by using Timer interrupts
- The PWM signal is used to turn ON and OFF a LED on the board itself.
- The PWM characteristics (frequency, duty cycle) can be configured with commands from the PC.
- The frequency can have 3 values (fast, medium, slow), and the duty cycle can have 3 values (25%, 50%, 75%).
- Additionally, the user can send a command for manual frequency configuration. After this command, the user will press the user button for a given amount of time, that will become the new frequency of the PWM signal.

1.2.3 Audio signal generation

- The board will generate an audible sinusoidal audio signal, using the audio codec and the DMA in circular buffer mode.
- The frequency of the audio signal is proportional to the frequency of the PWM LED signal, but in order to make it audible, it needs to be scaled (for example 1 Hz of PWM signal corresponds to 1 KHz of audio signal)
- Any changes to the PWM signal frequency should affect the audio signal as well
- The audio signal can be muted / un-muted.

1.2.4 Leds management

The discovery board has 4 user LEDs that will have different functions:

- LED_BLINK: This LED can have two modes of operation, that are selected with a command from the PC:
  - In PWM mode, the LED_BLINK changes according to the PWM signal generated internally
  - In manual mode the LED_BLINK can be turned ON and OFF with commands from the PC, and toggled with the user button.
- LED_ACCELEROMETERS: The remaining 3 LEDs will be used to indicate the readings from the accelerometer

1.2.5 Accelerometer

The accelerometer returns a value for each of the 3 axis (X, Y, and Z), and these values will be used to control the LED_ACCELEROMETERS

- The accelerometer will be read every 10ms. For this you can use another timer configured to generate interrupts.
- From commands, it is possible to enable or disable the accelerometer functionalities.
- If the accelerometer is disabled, the timer should be stopped and all LEDs turned OFF.
- If the accelerometer is enabled, after each reading one of the 3 LED_ACCELEROMETERS is turned ON, and the rest are turned OFF, according to which direction (X, Y, Z) as the highest accelerometer value, in absolute value.

1.2.6 Low power modes

- The Board will be in sleep mode as much as possible, to reduce power consumption.
- Additionally, the board can be sent to stop or standby mode by using a command from the PC.
- To wake up from stop mode, the user can press the user button on the Board.
- To wake up from standby mode, the user can press the reset button on the Board.

Command	Description
`stop`	Put Board in stop mode
`standby`	Put Board in standby mode
`changefreq`	Set the PWM frequency to the next possible value
`changedut`	Set the PWM duty cycle to the next possible value
`pwmman`	Set PWM frequency with user button
`ledpwm`	Set the LED_BLINK to PWM mode
`ledman`	Set the LED_BLINK to manual mode
`ledon`	Turn ON the LED_BLINK
`ledoff`	Turn OFF the LED_BLINK
`accon`	Enable accelerometer readings
`accoff`	Disable accelerometer readings
`mute`	Mute audible signal
`unmute`	Un-mute audible signal

Table 1: Commands. Students can change the command name if they wish, and add new commands.

A template for the project will be provided, with some of the basic functionalities already implemented. The student will have to implement the rest of the functionalities.

1.3 Multiple interrupts and low power modes: Callbacks, flags and the main loop

For our implementation, we will follow the callback, flags and main loop approach that consists of the following:

- In each interrupt callback, no action is performed other than setting a flag to indicate that the interrupt has occurred.
- In the main loop we check for the active flags and perform the actual action.

This is a very common approach in embedded systems, and it is very useful when we have multiple interrupts that need to be handled. It is always a good practice to do as little as possible in the interrupt handlers and perform most of the work in the main loop context.

In the main loop, we do not want to continuously check for the value of the flags, because this would be a waste of CPU time. Instead, we will use the **WFI** (Wait For Interrupt) instruction to put the CPU in a low-power state until an interrupt occurs.

Our platform supports 3 low-power modes, summarized here in order of decreasing power consumption:

1. **Sleep**: The Cortex-M4 is stopped, but all peripherals are still active. This means any peripheral interrupt enabled in the NVIC can wake up the device from Sleep mode.
2. **Stop**: All clocks and the 1.2V domain are stopped, therefore Cortex-M4 and peripherals are stopped. SRAM and registers are preserved. Only EXTI interrupts can wake up the device from Stop mode. (Blue button on the board).
3. **Standby**: Based on the Cortex-M4 deep sleep mode. All clocks are stopped, and the 1.2V domain is powered off. Cortex-M4 and CPU peripherals are stopped, and the SRAM and registers are lost. To wakeup from Standby mode, we use the Reset button on the board. As the SRAM and registers are lost, waking up from standby mode is like starting the firmware from scratch.

For more details, consult the hal documentation [1], section 51: HAL PWR Generic Driver.
A stripped-down version of a main file is shown in 1. Some important points to note:

- In lines 1 and 2, we define the values to indicate which interrupt has occurred. Note we use only one bit for each interrupt.
- In line 4, we define a variable to store the status of all interrupts, one bit for each interrupt, 1 means interrupt present. This variable is **volatile** because it is modified in the interrupt handlers and the main loop
- In lines 34 to 48 we have the callbacks for each interrupt, that just set the corresponding bit in the `isr_flags` variable, using the `bitwise OR` operator.
- In lines 13 to 16 we check that there are no interrupts pending and we put the CPU to sleep using the HAL function `HAL_PWR_EnterSLEEPMode`. Note we are using the `WFI` option so the CPU will be put to sleep until an interrupt occurs.
- In lines 20 to 29 we check which bit has been set in the `isr_flags` variable and perform the corresponding action. Note that we use the `bitwise AND` operator to check if a bit is set, and clear the bit using the `bitwise AND NOT` operator.
- We use two functions `handle_new_line` and `handle_timer10` to perform the actual action. This is just to keep the main loop clean and easy to read.

Get familiar with how the code works, and how the interrupts are handled. In the provided `main.c` file, we have added some comments to help you understand the code. Check how the stop is implemented and how it differs from the sleep mode.

1.4 PWM generation and elapsed time measurement

1.4.1 Generating the PWM signal

We can use a very simple approach to generate the PWM signal:

- We will use a timer to generate an interrupt every time the counter reaches the value in the `ARR` register.
- In the interrupt handler, we will toggle the state of the LED_BLINK, and will change the value of the `ARR` register.
- By alternating the `ARR` register between two values every time we get the interrupt, we will be able to generate a PWM signal.
- Depending on the ratio between the two values, we will be able to change the duty cycle of the PWM signal.

1.4.2 Measuring the button press time

We can use the EXTI interrupt and another timer to measure the time the user holds the button:

- When the EXTI interrupt is triggered, we will start a timer.
- The next time the EXTI interrupt is triggered, we will read the value of the timer counter
- Knowing the counting frequency of the timer, and the value of the counter we can calculate the elapsed time.

NOTE: We are not considering what happens if the counter reaches the ARR value and rolls over, but you should configure the prescaler so that you can measure reasonable time intervals. If you want you can also keep a counter of how many times the roll over happens and with this you can also address this corner case.

```c
const uint8_t ISR_FLAG_RX    = 0x01;  // Received data
const uint8_t ISR_FLAG_TIM10 = 0x02;  // Timer 10 Period elapsed

volatile uint8_t isr_flags = 0;

int main(void)
{
  // Initialization stuff..

  while (1)
  {
    // We check if that there are NO interrupts pending before going into sleep
    if (isr_flags == 0)
    {
      // Go to sleep, waiting for interrupt (WFI).
      HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI);
    }

    // After waking up, check what is the interrupt that woke us up. We use if clauses instead of if-else as more than one interrupt could be present at the same time
    if (isr_flags & ISR_FLAG_TIM10)
    {
      isr_flags &= ~ISR_FLAG_TIM10;
      handle_timer10();
    }

    if (isr_flags & ISR_FLAG_RX)
    {
      isr_flags &= ~ISR_FLAG_RX;
      handle_new_line();
    }
  }
}

void CDC_ReceiveCallBack(uint8_t *buf, uint32_t len)
{
  ...
  ...
  isr_flags |= ISR_FLAG_RX;
  ...
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == htim10.Instance)
  {
    isr_flags |= ISR_FLAG_TIM10;
  }
}

void handle_new_line(void)
{
  // Do something with the received data
}

void handle_timer10(void)
{
  // Do something when timer 10 expires
}
```

Listing 1: **main.c** main file example

1.5 Serial communication with host PC

Unfortunately, the stm32f4discovery does not expose any of the UARTs via the USB connections, so in order to implement serial communication with the PC, we rely on the USB_OTG configured as device, with the middleware USB_DEVICE configured as *Communication Device Class*. This allows us to communicate with the board as if it was a serial connection (in the same way we did in the previous lab with the UART). You can read more about CDC configuration in [2], but in the given material we already give you a basic working example.

The core of the working example is the callback in 2. This callback is invoked in interrupt context, every time new data is received via usb from the host, and accumulates the received data in a buffer until a new line is detected. When this happens, the accumulated potential command is copied to a buffer accessible by the main thread, and the corresponding interrupt flag is set (lines 35-36).

This callback is inspired in the answer in [3].

```
1  void CDC_ReceiveCallBack(uint8_t *buf, uint32_t len)
2  {
3    // Prevent overflow, does not handle the command
4    if (line_len + len >= LINE_BUFFER_SIZE)
5    {
6      line_len = 0;
7      return;
8    }
9
10   // Append received chunk
11   memcpy(&line_buffer[line_len], buf, len);
12   line_len += len;
13
14   // Process all complete lines
15   while (1)
16   {
17     // Look for '\n' or '\r' inside line_buffer
18     uint8_t *line_feed = memchr(line_buffer, '\n', line_len);
19     if (line_feed == NULL){
20       line_feed = memchr(line_buffer, '\r', line_len);
21       if (line_feed == NULL)
22       break;
23     }
24
25     // Replace \n or \r by terminator
26     *line_feed = '\0';
27
28     // Remove optional '\r' in case input was \r\n
29     if (line_feed > line_buffer && *(line_feed - 1) == '\r')
30     *(line_feed - 1) = '\0';
31
32     uint32_t processed = (line_feed + 1) - line_buffer;
33
34     // Signal the main that there is a new line ready to be checked
35     memcpy(line_ready_buffer, line_buffer, processed);
36     isr_flags |= ISR_FLAG_RX;
37
38     // Move leftover bytes to start
39     line_len -= processed;
40     memmove(line_buffer, line_buffer + processed, line_len);
41   }
42 }
```

Listing 2: Detecting lines from the CDC callback

1.6 Audio playback

For audio playback, we will use the audio codec device present in the discovery board. We will use the project in [4] as basis. This project shows how to configure the audio codec using I2C, and how to stream

audio to it using the DMA to hold the data to be played in a circular buffer. The DMA is connected to the codec via I2S.

The project also provides a simple driver that you can use to

- Initialize
- Start playback
- Stop playback
- Mute
- Un mute

In the provided project you will find the I2S, DMA and I2C already configured, as well as an example in the main of how to initialize and create a simple audible sinusoidal signal.

1.7 Accelerometer

For reading values from the accelerometer, we will use the BSP library by ST for the stm32f4discovery board. This BSP library contains some high level function to control components present in the board. To make compilation easier, in the provided project we include the source / header related to the accelerometer library: **stm32f4_discovery_accelerometer**.

You can use the APIs in the file to interact with the accelerometer. For an example of how to use it, you can refer to the ST demonstration project for the discovery board, in [5], that shows how to use the accelerometer (among other stuff).

NOTE: in the demonstration they use the **systick** interrupts to read the accelerometer, changing the **systick** period to 10ms. To avoid issues by changing the systick period, please use a regular timer to generate the interrupts, configured to 10ms.

2 Homework

The provided project is not complete and does not meet the requirements described in the first section. Your job is to understand what has been done so far, and complete the project.

You are free to improve the provided code or add more functionalities.

For any modification that you perform, please add a comment explaining what the modification does.

We suggest to download the provided project, and initialize a git repository. Then commit the original status of the project. This will be useful for two things:

- During support and evaluation, it will make it very easy to understand what are the modifications that you have done to the project.
- When using **STM32CubeMX** and **stm32pio**, remember that the autogenerated files will be overwritten, and only the sections marked with **/*USER CODE .. */** are kept, we suggest to commit your modifications before you make any changes with **stm32pio**. In this way:
  - You can understand what modifications have been made to the firmware. For example, if you add a new timer, what files / changes are done.
  - If by mistake you write code outside the **/*USER CODE .. */** sections, you can recover it by looking at the diff with vscode.

References

[1] STMicroelectronics. um1725 description of stm32f4 hal and lowlayer drivers. website, . URL https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf.

[2] Will Frank. Stm32: Usb virtual com port (vcp). website. URL https://willfrank.co.uk/stm32-vcp.html.

[3] Codo. Answer to: stm32 usb receive more than 1 byte. website. URL https://stackoverflow.com/a/59982031.

[4] Yerkebulan. Stm32 audio codec. website. URL https://hackaday.io/project/195461-stm32-audio-codec.

[5] STMicroelectronics. Stm32 audio codec. website, . URL https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/STM32F4-Discovery/Demonstrations.

[6] Walter Gallego. Best practices: How to work with interrupts and low power modes when using the hal. website. URL https://community.st.com/t5/stm32-mcus-products/best-practices-how-to-work-with-interrupts-and-low-power-modes/m-p/620588#M230333.

[7] Gabriel Staples. Which variable types/sizes are atomic on stm32 microcontrollers? website, . URL https://stackoverflow.com/questions/52784613/which-variable-types-sizes-are-atomic-on-stm32-microcontrollers.

[8] Gabriel Staples. Reading a 64 bit variable that is updated by an isr. website, . URL https://stackoverflow.com/questions/71624109/reading-a-64-bit-variable-that-is-updated-by-an-isr/71625693#71625693.

[9] Gabriel Staples. What are the various ways to disable and re-enable interrupts in stm32 microcontrollers in order to implement atomic access guards. website, . URL https://stackoverflow.com/questions/71626597/what-are-the-various-ways-to-disable-and-re-enable-interrupts-in-stm32-microcont/71626598#71626598.

[10] Majerle Tilen. How to properly enable/disable interrupts in arm cortex-m. website. URL https://stm32f4-discovery.net/2015/06/how-to-properly-enabledisable-interrupts-in-arm-cortex-m/.

```
1  while (1)
2  {
3      // Disable CPU from running interrupts. This means that if an interrupt
       arrives, it will still be added to the NVIC pending list, but the callbacks
       will not be invoked.
4      __disable_irq();
5
6      // We check if that there are NO interrupts pending before going into sleep
7      if (isr_flags == 0)
8      {
9          // Go to sleep, waiting for interrupt (WFI).
10         HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI);
11         __enable_irq();
12     }
13     else
14     {
15         // Select the task to execute
16         // Clear the active flag for the selected task
17         __enable_irq();
18         // Execute the selected task
19     }
```

Listing 3: Interrupt disabling

Appendix A Interrupts and race conditions

This section is not required for the development of this laboratory. However, we leave it here for completeness on the topic of interrupts.

When working with interrupts in projects that require high reliability, it is common to improve on the approach that we described before, by making sure there are no possible race conditions in the flag handling. Consider the code in 1. What happens if an interrupt arrives after we have checked the flags in line 13, but before we go to sleep on line 16?. The Cortex will enter sleep mode and the interrupt will not wake it up. The interrupt will go unserved until another interrupt arrives to wake up the CPU.

To avoid this situation, interrupt disabling at CPU level can be used. Interrupts can be disabled at 3 levels:

- **peripheral:** Each peripheral can be configured to generate or not an interrupt. Done in the peripheral registers or HALs.
- **NVIC:** For an interrupt to be enabled it has to be enabled in the NVIC. Done with the HAL functions `HAL_NVIC_EnableIRQ()` and `HAL_NVIC_DisableIRQ()`.
- **CPU:** The CPU can be configured to run or not interrupts. Done with the ARM functions `__enable_irq()` and `__disable_irq()`.

ARM CPUs wake up from sleep when the interrupt becomes pending in the NVIC, regardless of whether the interrupt is enabled at CPU level. Thanks to this we can use the approach in 3:

- If an interrupt arrives before line 4, it will set the flag and the CPU will not go to sleep.
- If an interrupt arrives after line 4, it will not set the flag but in line 10 the CPU will not even go to sleep, because the interrupt is pending in the NVIC. In line 11, after reenabling the interrupts, the callback will be executed, the flag set and the interrupt served in the next cycle.

Besides guarding the flag handling in the main loop, it is also required to do it in each callback. As you can see this topic is quite complex, especially when working with bare metal and not using an RTOS. If you are curious, some interesting discussions can be found in [6], [7], [8], [9], [10].

Appendix B Timer channels

This section is not required for the development of this laboratory. However, we leave it here for completeness on the topic of timers.

STM32 timers provide other functionalities besides interrupt generation. See for example the timer10 of our stm32f4 board in CubeMx, Figure 2. In Channel1 it can perform various actions, including *Input Capture*, *Output Compare* and *PWM generation*. The channel has an associated GPIO, which in this case is PB8, that is configured as *Alternate Function* when the channel is enabled.

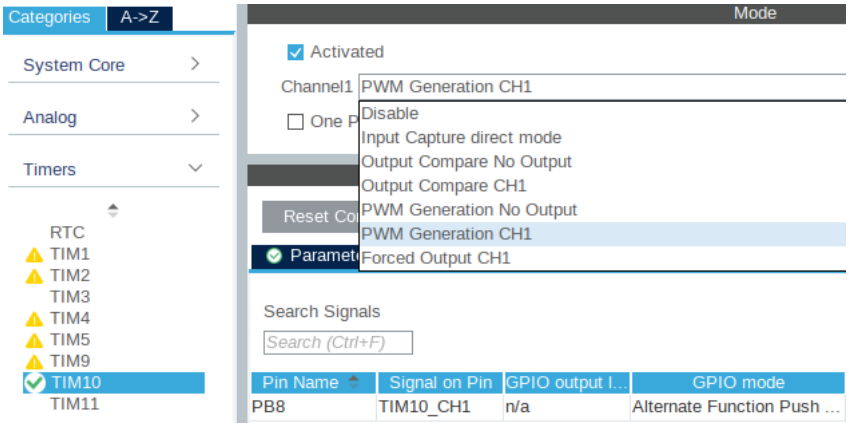

Figure 2: Timer 10 channel configuration

Using channels should be the preferred approach when implementing tasks such as PWM generation or time interval measurement, as the task can be carried out by the timer independently, without the need for intervention from the CPU.