Politecnico di Torino

Digital Systems Electronics

04OIHNX - A.Y. 2024/2025

Laboratory assignment n°. 10:

Input capture and Pulse Width Modulator

Due date: 03/06/2025 Delivery date: 03/06/2025

Group 08 - Contributions:

310779, Simone Viola - 33,3%
300061, Viorel Ionut Bohotici - 33,3%
308299, Daniele Becchero - 33,3%

The members of the group listed above declare under their own responsibility that no part of this document has been copied from other documents and that the associated code is original and has been developed expressly for the assigned project.

Contents

1 Measure of the frequency and duty cycle of an external square waveform

This section is about the design of a MCU application capable of measuring the frequency ans the duty cycle of a square waveform. The measurement scheme is based on a sort of indirect frequency counter relying on the input-capture feature of STM32 timers.

1.1 Application Description

Channel 1 of TIM3 is configured as an input compare pin to sample the counter value upon each edge transition—both rising and falling—allowing measurement of both frequency and duty cycle, which are evaluated starting from the value in the counter register at each edge. The rising edges of the signal are used to evaluate the period, while falling edges are needed to compute the high period, and consequently the duty-cycle. Each time an edge is detected, an interrupt request is triggered, and the computation of the measured parameters is executed within the ISR. If an update event occurs in the counter of TIM3, the overflow condition is considered to ensure correct computation of the frequency and duty-cycle. The measurement results are transmitted via UART every 500 ms.

1.2 Code Implementation

Starting from a new project in CubeMX, the needed peripherals have been initialized. `TIM3` clock has been configured as "internal clock" and CC channel 1 is set as "Input compare direct mode", which highlights pin `PA6` of the MCU in the MCU view. In the channel 1's parameter settings, the polarity has been configured as "Both edges" as discussed above, while in NVIC settings, global interrupt of TIM3 has been enabled. Priority group for NVIC has been configured according to instructions given in the document attached of the previous laboratory session. The CubeMX-generated code utilizes LL drivers for all peripherals, except for USART2, which relies on HAL drivers and is used to transmit result data to another properly configured device.

In `main.c`, two volatile variables have been defined in the appropriate section: they store the current period and duty-cycle.

```
volatile uint16_t period = 0;
volatile float dutyCycle = 0;
```

Then, before the infinite loop, the configuration code configures the timer and initializes a constant that stores the rate at with `TIM3_CNT` updates its content. In details, for TIM3

- Prescaler has been set to 99 (timer clock of 840 kHz).
- Auto-reload value has been set to 0xFFFF.
- Interrupt for CC1 and update event has been enabled.
- Capture/Compare channel 1 has been activated.

Before entering the application code, the timer is started and it is configured to freeze in debug mode to ensure proper operation within debug.

The application code relies on a set of instructions that compute the frequency and the duty cycle with data coming from other variables - edited within ISRs - and a set of instructions to send the measurement result over UART every 500 ms.

The real core of the applications lies in the ISR of `TIM3`, which is where all the critical computations are done. The ISR is located in file `stm32f4xx_it.c`. Several auxiliary variables have been initialized to facilitate the measurement process. They are configured as static, ensuring their values persist across function calls. Also variables defined in `main.c` have been declared, but this time as an extern variable, meaning that those variables are shared between different files.

```
static uint16_t prevInterval = 0;   // Previous interval
static uint16_t interval = 0;       // Current interval
static uint16_t highTime = 0;       // HIGH interval
static uint8_t ovf_cnt = 0;         // CNT overflow counter

extern uint16_t period;             // Period of the waveform
extern float dutyCycle;             // DC of the waveform
```

Finally, `TIM3_IRQHandler` is where the computation is done. Within ISR, the first operation to be done is to know who requested the interrupt. If it has been raised by CC1IF, the captured value is stored and the flag is then cleared. Then, to understand the type of edge - falling or rising - the logical value of `PA6` is red. If its value is `HIGH`, this means that a transition from low to high had occurred. If so, the period of the waveform is computed with the following formula. Then, `prevInterval` is updated with the current value and `ovf_cnt` is cleared.

$$\text{Period} = ((\text{current CNT} + \#\text{ of CNT overflows} \cdot \max(\text{CNT})) - \text{previous CNT})$$

If the value of `PA6` is `LOW`, then a falling edge is detected and `highTime` value is computed with the formula already seen for the period. Notice that an interrupt may be requested also by an update event, so that situation must be handled too. The proper flag is checked, and if its value is `HIGH`, variable `ovf_cnt` is incremented by one and the flag is then cleared. At the end of the ISR, the new value of DC (expressed as a percentage) is computed with the formula

$$\text{dutyCycle} = \frac{\text{highTime}}{\text{period}} \cdot 100.0$$

1.3 Verification and testing

The application has been successfully tested in laboratory. The setup relies on the arbitrary function generator set to generate a square wave with programmable frequency and duty cycle and connected both to the oscilloscope and to pin `PA6` of the MCU - which corresponds on Arduino connector `D12` on the Nucleo board. The AFG must be configured accordingly to avoid damaging the board with too high voltages applied to its

pins. We have had some issues when trying to run freely the application in debug mode without any enabled breakpoint, so we decided to use the UART transmission between the MCU and another PC. Running Visual Studio Code, with extension named "Serial Monitor" installed, we started a serial monitor instance with the a baud rate of 115200 and 8 bits word length (default parameters previously set in CubeMX) and we were able to see that the computed frequency and duty cycle matches the values set on the AFG. The precision of the duty cycle measurement is higher than that of the frequency, which appears to be affected by an instrumental error that causes a constant deviation between the value measured with the scope and with STM32.

Using breakpoints in debug mode and periodically restarting execution resulted in incorrect measurement values. This behavior cost us significant time as we tried to identify the source of the issue. However, after exiting debug mode and allowing the MCU to run freely, the application functioned as expected. This suggests that using a debugger (with breakpoints) in time-sensitive operations can lead to inaccurate results, whereas the same code may work correctly when the MCU is allowed to run without debug interruptions.

The theoretical upper-bound for the frequency is half the timer frequency (420 kHz, due to the Nyquist sampling theorem. The lowest detectable frequency without any overflow event is given by

$$f_{\text{min}} = \frac{f_{\text{timer}}}{65535} \approx 12.8 \text{ Hz}$$

By handling the overflow condition, including an overflow counter that allocates 8 bits, the new lowest frequency is

$$f_{\text{min,new}} = \frac{f_{\text{timer}}}{65535 \times 255} \approx \frac{840 \text{ kHz}}{16.7 \times 10^6} \approx 0.05 \text{ Hz}$$

The computed frequency limits are overestimated, as the actual operational timing of each instruction is unpredictable. These calculations should take into account the execution time of the entire application, as various factors—such as interrupt latency, instruction processing time, and system load—can impact measurement accuracy and limits. The higher limit is set by the execution time of the longest sequence of functions, which must not exceed the 100 clock cycles (since timer clock is 100 times slower that system clock). These limits may be extended by optimizing the code execution or changing the value of the prescaler and/or system clock and the space allocated for the overflow counter. An increase in the timer clock also leads to a better accuracy, resulting in a higher timer resolution and a shorter timer period. Consequently, the lowest measurable frequency increases, as indicated by the formula above. Another potential enhancement involves upgrading the external clock reference. Replacing the internal RC oscillator with the HSE crystal oscillator—or upgrading to a TCXO or OCXO—may significantly improve stability and performance.

2 Pulse Width Modulation

2.1 Square wave generation

This section is about a square wave generator with fixed frequency and variable duty-cycle proportional to an input signal's frequency.

2.1.1 Application Description

The application is based on the previous section. The input signal must operate within a frequency range of 800 Hz to 12 kHz, with the corresponding output signal's duty cycle varying proportionally between 25% and 75%. Specifically, a frequency of 800 Hz results in a 25% duty cycle, whereas 12 kHz leads to a 75% duty cycle. The output signal is generated on pin `PB6` relying on PWM capability of STM32's timers.

2.1.2 Code Implementation

In CubeMX, the configuration started with the configuration of peripherals described in Section 1. Additionally, `TIM4` is enabled and driven by the internal clock. Its capture/compare channel 1, which is the alternate function of GPIOB, pin 6, is configured as "PWM mode 1", which set the timer to automatically generate a PWM signal based on the content of registers `ARR` and `CCR1`. The frequency of the waveform is defined by both `PSC` and `ARR` (see formula 1), while the duty-cycle is defined by the content of register `CCR1`. DC is defined as the `HIGH` period of the signal over the entire period. In PWM mode 1, it is determined by the ratio of `CCR1` to `ARR`, as expressed in Formula 2. Since the DC value is dependent on the frequency of the external waveform, the value for `CCR1` has not been set because it will change from time to time.

$$\frac{\text{Timer clock}}{(1+\text{PSC})(1+\text{ARR})} \tag{1}$$

$$\text{DC} = \frac{1+\text{CCR1}}{1+\text{ARR}} \tag{2}$$

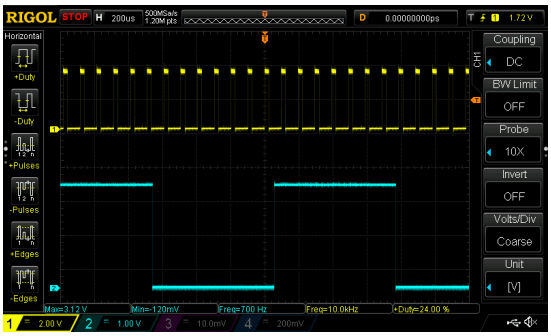
The tool generates all the files for this project. The configuration code for `TIM3` and its associated ISR are equal to that of the previous exercise. In addition to that, the configuration of `TIM4` is done as usual. To achieve an output frequency of 10 kHz starting from a timer clock of 84 MHz, `PSC` has been set to 99 and `ARR` to 99, according to Formula 1. `CCR1` is initialized at half of its maximum value, meaning 50% DC, and then CC channel 1 is enabled. Then, both timers are started and set to freeze their behaviour in debug mode, while the execution is stopped. Before entering the application code, a constant is evaluated to speed-up the computations. Before entering the description of the application code, the ISR part is brifely described. The same functions of the previous exercise have been included in `stm32f4xx_it.c`-apart from the duty cycle computation-since the entire application relies on many frequency measurement. One major difference lies in the ISR of `TIM3`: now, the period (in number of pulses) is computed within the ISR, and a flag is set. It is used as the condition to execute the conversion of the pulse value into a significant time interval, but this time it is computed in the application code (hence, not

during an ISR). Since the conversion involves a division operator (which is highly time and resource-consuming), this change would reduce significantly the time spent within an ISR while computing the new frequency value only when needed. No other ISRs are needed for this application.

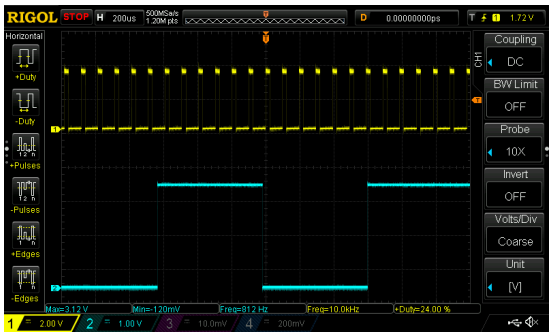
Back to `main.c`, the application code is located within the infinite loop. First, the system checks if the period measurement has ended. If true, the frequency is computed and register `CCR1` of `TIM3` is updated when the frequency falls within the two bounds given as specification. Notice that the frequency is computed as a floating-point operation, but the result is defined as an integer because there is no need of very high resolution in frequency. The value of output signal's duty cycle, which is related to the `TIM4_CCR1`, is computed using a linear interpolation and stored into a 16-bit unsigned variable to match the dimension of the timer's register. Then the value of duty cycle is converted into an appropriate value for `CCR1` and loaded. Notice that, in this application, the last conversion is not needed because `ARR` is already limited to 99 (meaning 100 different values), and the duty cycle (expressed as a percentage) is limited by definition by the same bounds. Before exiting the outer conditional statement, the variable `updatedValue` is cleared.

2.1.3 Verification and testing

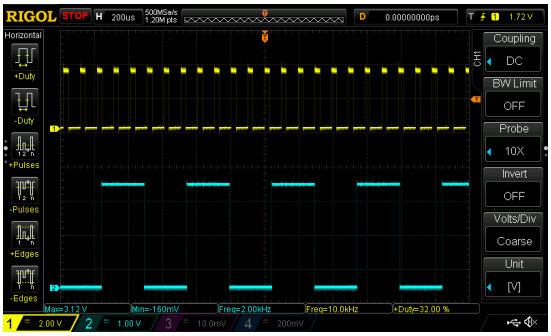
The setup to test this application involves an oscilloscope and an AFG. The signal generated by the function generator is connected to both pin `PA6` of the MCU and channel 2 of the oscilloscope, represented by the cyan trace. In addition to that, pin `PB6` is connected to a probe, which is then routed to channel 1 of the oscilloscope, represented by the yellow trace. Few essential measurements are computed automatically by the scope and they are visible in the lower part of the following pictures. The input frequency is swept manually in and out of the bounds.



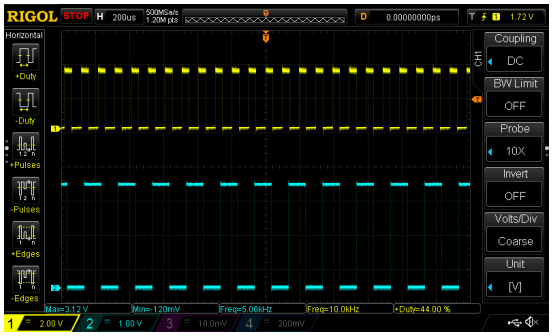
(a) $f_{\text{in}} = 700\,\text{Hz}, \text{DC}_{\text{out}} = 24\%$



(b) $f_{\text{in}} = 812\,\text{Hz}, \text{DC}_{\text{out}} = 24\%$



(c) $f_{\text{in}} = 2\,\text{kHz}, \text{DC}_{\text{out}} = 32\%$



(d) $f_{\text{in}} = 5\,\text{kHz}, \text{DC}_{\text{out}} = 44\%$

Figure 1: Project tested in lab

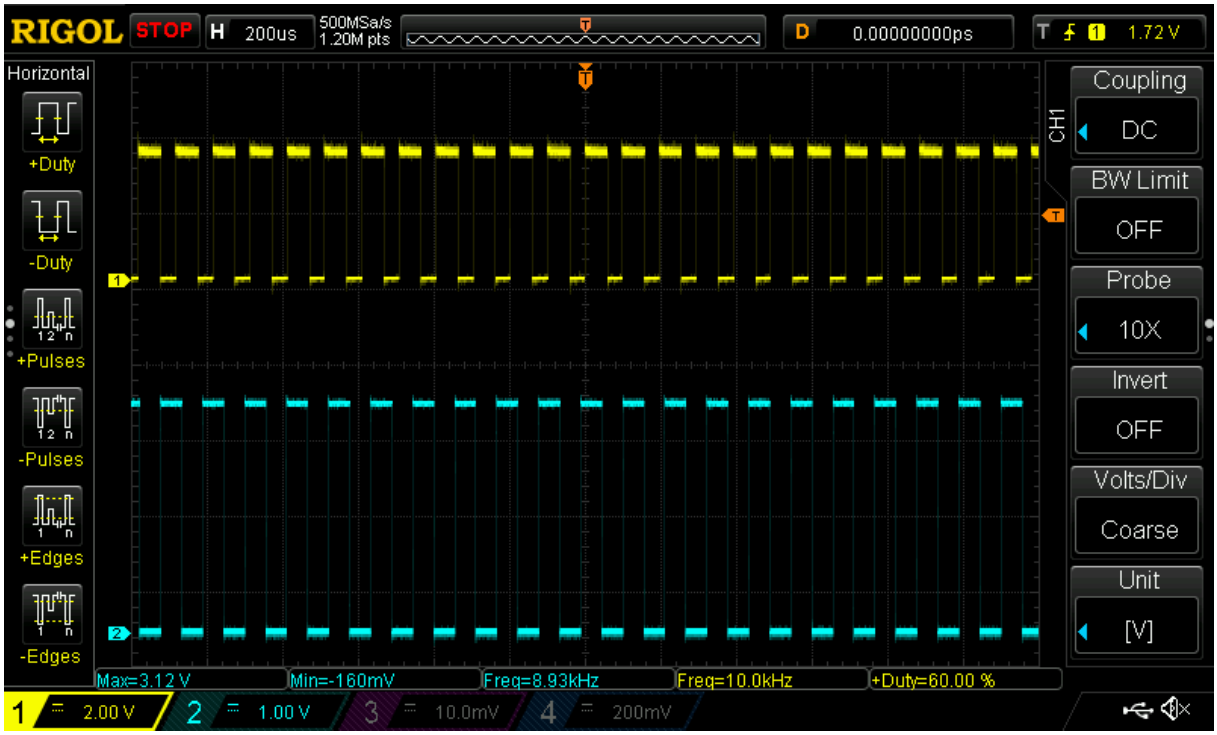


Figure 2: $f_{\text{in}} = 8.93\,\text{kHz}, \text{DC}_{\text{out}} = 60\%$

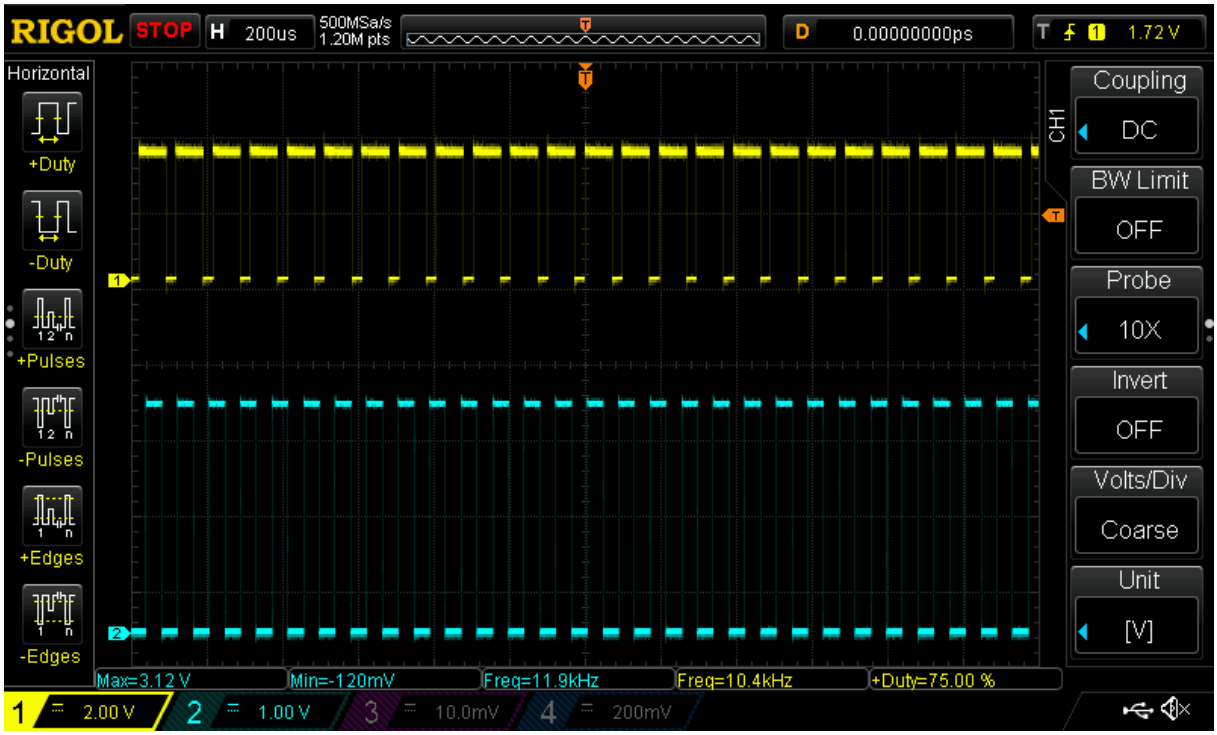


Figure 3: $f_{\text{in}} = 11.9\,\text{kHz}, \text{DC}_{\text{out}} = 75\%$

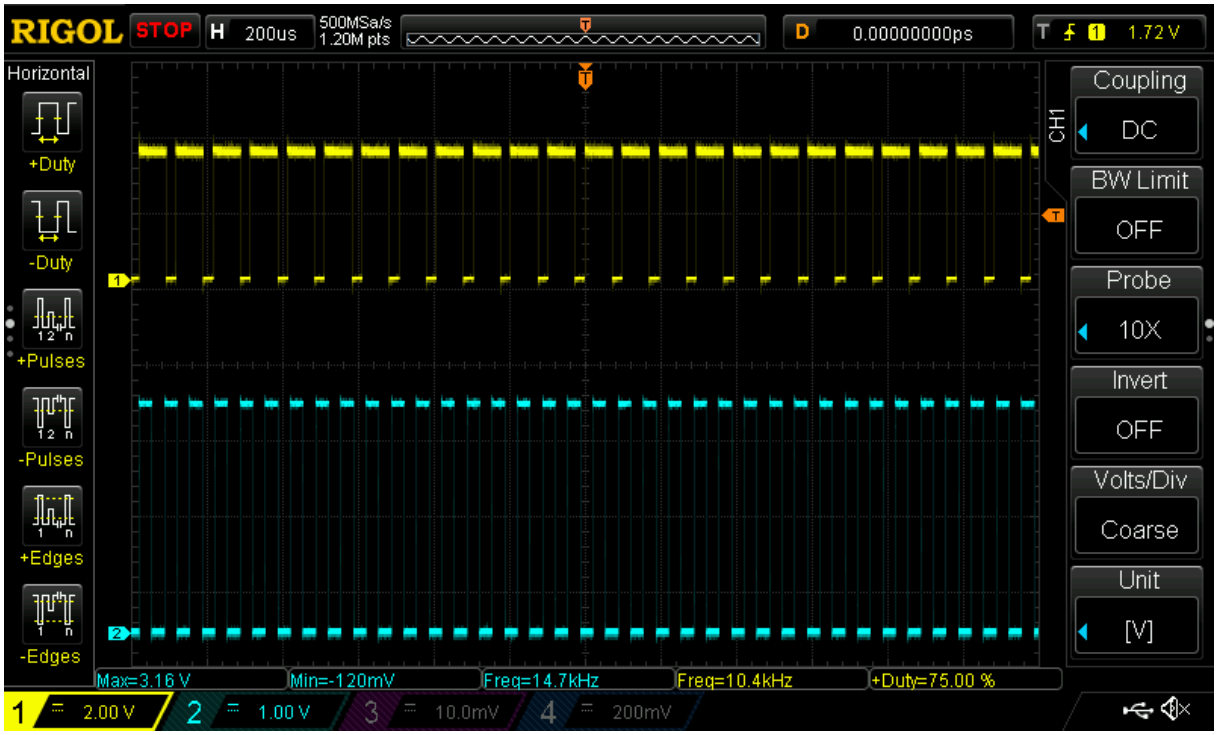


Figure 4: $f_{\text{in}} = 14.7\,\text{kHz}, \text{DC}_{\text{out}} = 75\%$

2.2 LED Dimmer

After gaining practical experience in generating square waves with specific frequencies and duty cycles, these skills can be applied to control the brightness of an LED using Pulse Width Modulation (PWM). This technique adjusts the effective power delivered to a load by varying the duty cycle of the supply voltage.

2.2.1 Application Description

In this application, PWM is used to control the brightness of the on-board LED2 by gradually increasing the duty cycle from its minimum to maximum value (fade-in), and then decreasing it back to the minimum (fade-out). The implementation relies on two `for` loops inside an infinite `while(1)` loop. The first `for` loop handles the fade-in phase by incrementing the duty cycle from 0% to 100%, while the second `for` loop handles the fade-out phase by decrementing the duty cycle from 100% back to 0%. This cycle repeats continuously.

To configure the Nucleo board in the `.ioc` file, `Channel1` of `TIM2` is set as "PWM Generation CH1" to provide the PWM output to the PA5 pin. Then, in the `TIM2 GPIO Settings`, the signal on `PA5` is linked to `LED2`, allowing control of the on-board LED and verification of the correct implementation of the code.

The key component of this implementation is the configuration of TIM2, which defines the characteristics of the PWM signal. Given the clock settings, TIM2 operates at 84 MHz. To simplify timing calculations, a prescaler is selected to produce a timer clock of 100 kHz, meaning each timer tick corresponds to 1 µs. The PWM frequency is set to 1 kHz, ensuring smooth brightness transitions and eliminating visible flickering.

2.2.2 Code Implementation

Once the configuration is complete, the corresponding initialization code is generated and supplemented with additional implementation. In the `main.c` file, we begin by initializing the function that enables output compare for the specified timer channel, starting PWM output.

```
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
int duty = 0;
```

We then declare an integer variable named `duty`, which is used to store and control the PWM duty cycle value, ranging from 0 to 100. Inside the `while(1)` infinite loop, there are two `for` loops: the first one performs the fade-in phase by incrementing `duty` from 0 to 100, while the second one performs the fade-out phase by decrementing it back from 100 to 0. As this is an infinite loop, the fade-in and fade-out phases repeat continuously.

```
while (1) {
    for (duty = 0; duty <= 100; duty++) {
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, duty);
        HAL_Delay(10);
    }
    for (duty = 100; duty >= 0; duty--) {
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, duty);
        HAL_Delay(10);
    }
}
```

In each of the two `for` loops, the macro `__HAL_TIM_SET_COMPARE` updates the compare value for the duty cycle of the specific channel `CH1` of timer `TIM2`. The `HAL_Delay` function determines the speed of the fade-in and fade-out transitions. Since the assignment requires a 1-second duration for each transition, and each fade loop contains 101 steps (from 0 to 100), setting a delay of 10 ms per step results in a total duration of approximately 1 second for each phase.

2.2.3 Verification and Testing

After compiling the code and flashing it onto the Nucleo board, the brightness variation of the on-board `LED2` can be observed, showing correct dimming without visible flickering. This indicates that PWM is implemented with an appropriate frequency. By observing the waveform on the LED pin using an oscilloscope, we can see that the duty cycle changes in a "triangular" pattern, as the two `for` loops increase and decrease the `duty` value linearly. As expected, the PWM output amplitude ranges from 0 to 3.3 V. Below are oscilloscope captures at three different duty cycle levels.

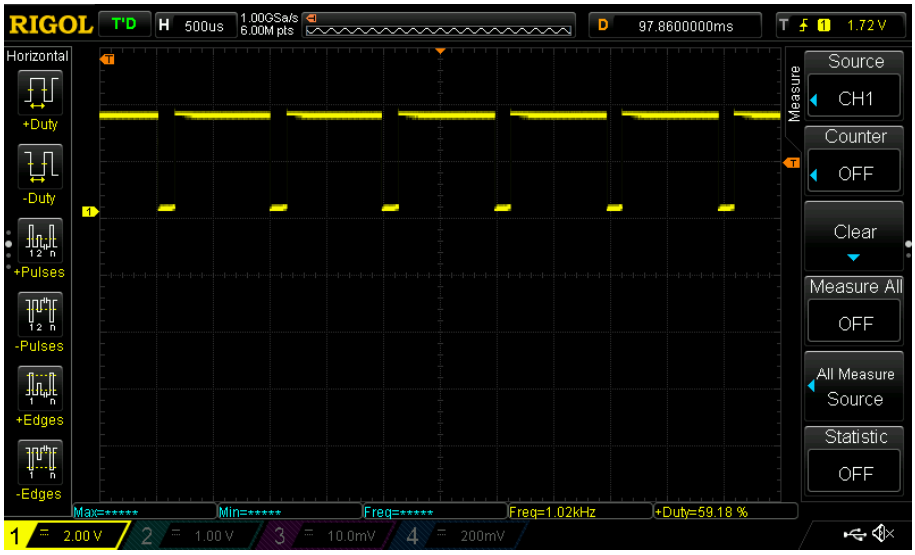


Figure 5: LED PWM waveform

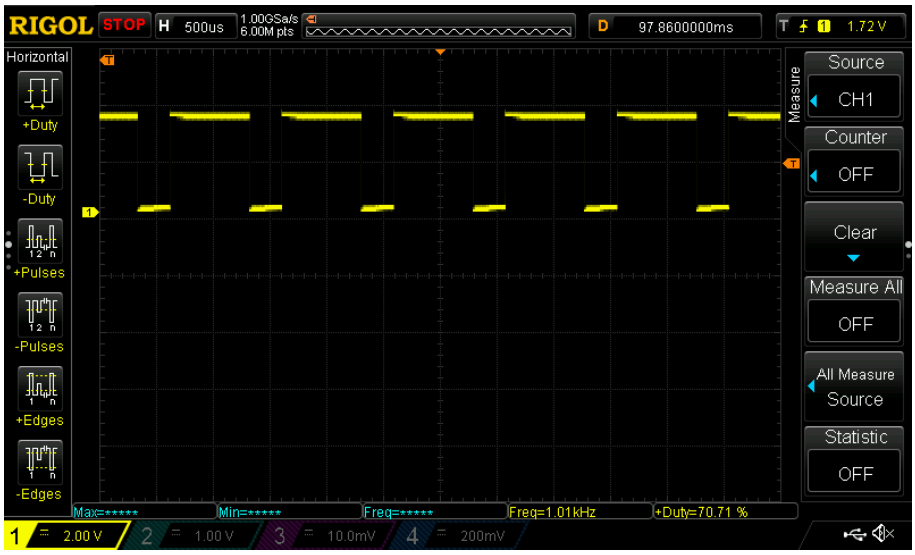


Figure 6: LED PWM waveform

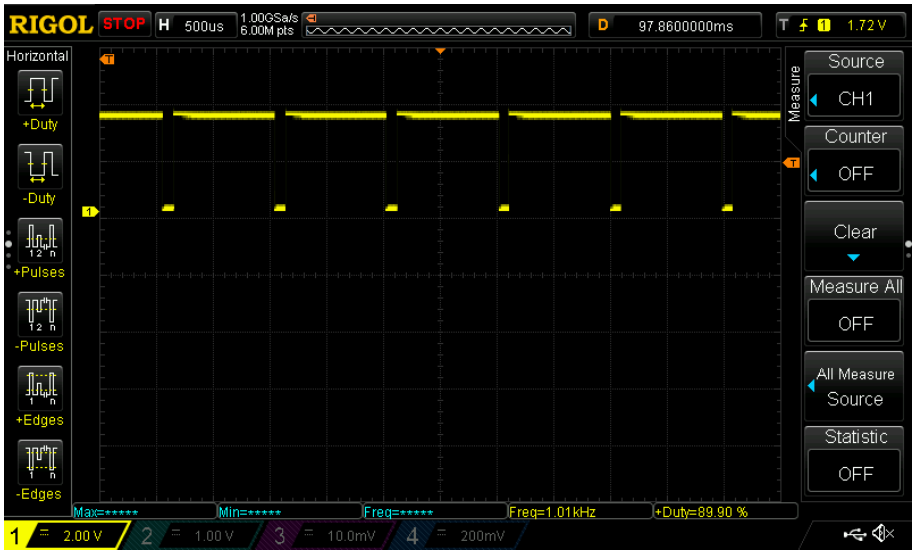


Figure 7: LED PWM waveform

Doubling the period and halving the prescaler results in the appearance of a "dead time" effect. This occurs because the ARR (Auto-Reload Register) is set to a maximum of 49, while `CCR1` (the compare register) varies from 0 to 100, exceeding the configured period and leading to unintended behavior. To ensure correct PWM operation, the code for the delay (created by using HAL) should be modified according to the PSC, ARR, and timer clock parameters.

2.3 Sine wave generation

In this section, a method for implementing a simple digital-to-analog converter (DAC) using PWM and DMA on a microcontroller is discussed.

2.3.1 Application Description

A 50 Hz sine waveform is generated by storing 200 duty-cycle values in a lookup table (`LUT`). These values are automatically transferred to the PWM register at each sampling interval using DMA, enabling continuous waveform output without CPU intervention. This approach illustrates an efficient and low-cost technique for DAC implementation. Finally, to better visualize the digitally generated sine wave converted to analog on the oscilloscope, a reconstruction filter based on a simple low-pass filter is added.

2.3.2 Code Implementation

The implementation of this project begins, as in the previous section, with CubeMX to configure the `.ioc` file. Based on these settings, the initial code is then generated. In this step, the `TIM3` timer is configured for `PWM` (Pulse Width Modulation) signal generation by enabling the `"Channel 1: PWM Generation CH1"` setting. This configuration allows `TIM3` to operate as a `PWM` generator, with the output signal routed to the `PA6` pin. Next, in the Parameter Settings, we configure the Prescaler to 83 and the Counter Period (Auto-Reload Register, `ARR`) to 99. As previously discussed in the report for Laboratory 9, setting the TIM3 prescaler to 83 results in 84 timer clock cycles (since counting starts from 0). Recall the formula:

$$\text{Timer Clock} = \frac{\text{System Clock}}{\text{PSC} + 1} = \frac{84\ \text{MHz}}{83 + 1}$$

Regarding the `ARR` value, since we are generating a `PWM` signal, setting `ARR` = 99 provides 100 discrete levels of resolution for the duty cycle (ranging from 0 to 99). This allows for precise and sufficiently accurate control of the PWM output. Moving on to the code, we start by including the standard library `math.h`, which we will later use for the `sinf` function to generate the sinusoid we intend to sample. Speaking of samples, we define a constant `SAMPLE_SIZE` with a value of 200 using `#define`, as specified in the assignment. This value determines the number of samples per period of the function. Naturally, increasing the number of samples would result in a more accurate representation of the sinusoid using `PWM`.

```
#include "main.h"
#include <math.h>
#define SAMPLE_SIZE 200
```

Next, in the `int main(void)` function, we initialize a float array with a size equal to the number of samples, previously set to 200. This array will store the sampled values of the sine wave. Then, a 16-bit unsigned integer Look-Up Table `SINE_LUT` is defined to store the quantized values suitable for PWM output. To generate the waveform, a `for` loop is used to compute the values of a sine wave sampled over a complete period.

The `for` loop iterates from 0 to `SAMPLE_SIZE - 1`. Within the loop, the sine function is evaluated at equidistant points using the expression `sinf(2*M_PI*i/SAMPLE_SIZE)`, which produces values in the range $[-1, 1]$. To ensure all values are positive, suitable for PWM generation, the result is shifted upwards by adding 1, transforming the range to $[0, 2]$. These normalized values are stored in the `sampledSine` array as floating-point numbers. Each value is then scaled and quantized to an integer range suitable for PWM duty-cycle control.

```
float sampledSine[SAMPLE_SIZE];
uint16_t SINE_LUT[SAMPLE_SIZE];

for(uint8_t i = 0; i < SAMPLE_SIZE; i++){
    sampledSine[i] = sinf(2*M_PI*i/SAMPLE_SIZE) + 1;
    SINE_LUT[i] = (sampledSine[i]*100/2);
}
HAL_TIM_PWM_Start_DMA(&htim3,TIM_CHANNEL_1,(uint32_t *)SINE_LUT,200);
```

Specifically, each value is multiplied by 100 and then divided by 2, effectively mapping the values to a range from 0 to 100. These quantized values are stored in the 16-bit unsigned integer LookUp-Table `SINE_LUT`. Finally, the function `HAL_TIM_PWM_Start_DMA` is called to start the PWM signal generation on timer channel `TIM_CHANNEL_1` using DMA. The DMA is configured to continuously stream the contents of the `SINE_LUT` array to the PWM hardware, ensuring real-time and efficient waveform output without CPU intervention.

2.3.3 Verification and testing

The setup used to test this application includes an oscilloscope, a breadboard, a resistor, and a capacitor. Initially, during the debugging phase, when we only need to verify the correct operation of the PWM, we can connect the oscilloscope probe directly to pin `PA6`, which carries the PWM output signal. The ground (GND) of the Nucleo F401 board should be connected to the probe's ground to ensure a proper reference during measurements.

After confirming that the PWM operates correctly at a frequency of 10 kHz, and that the duty cycle increases and decreases in a sinusoidal-like manner, we can introduce a reconstruction filter circuit to convert the digital PWM signal into a smoother analog representation of the output sine wave.

This filter is a basic RC low-pass circuit composed of a resistor of 220 Ω and a capacitor of 1 µF. The circuit attenuates high-frequency components, allowing the capacitor to charge during the high state of the PWM and discharge during the low state. The resulting output, shown in Figures 8 and 9 closely resembles the desired sine wave.
In particular, this low-pass filter is designed with a cutoff frequency given by the following formula:

$$f_c = \frac{1}{2\pi RC} \approx 723\,\text{Hz}$$

Since the desired frequency of the sine wave is 50 Hz and the `PWM` frequency is around

10 kHz, the cutoff frequency f_c is set slightly higher than the sine wave frequency. This allows the filter to effectively attenuate the high-frequency components caused by the sharp transitions of the digital PWM signal, while still providing a sufficiently wide passband to preserve the fundamental component of the sine wave, ensuring it is not significantly affected by the filtering. We can now use another channel of the oscilloscope to display also the board's output. This allows us to compare the `PWM` signal before and after filtering, verifying that the higher (or lower) peaks of the sine wave correspond to the maximum (or minimum) duty-cycle of the `PWM`. Each transition between maximum and minimum duty cycles is properly implemented to represent a sine wave, and the signals shown in the figures 8 and 9 serve as proof. In the figures below, the yellow signal is the PWM output from PA6 of the Nucleo Board, and the blue signal is the PWM after RC filtering, resulting a reasonably accurate sine wave. Note that the oscilloscope probes were set to 10X attenuation while both channels were set as 1X, so the vertical scale in picture is not correct. Anyway, to obtain the correct vertical scale, the value of a vertical division must be multiplied by 10. Based on this, we can confirm that the PWM signal amplitude ranges from 0 to 3.3 V, and the sine wave amplitude is consistent with expectations.

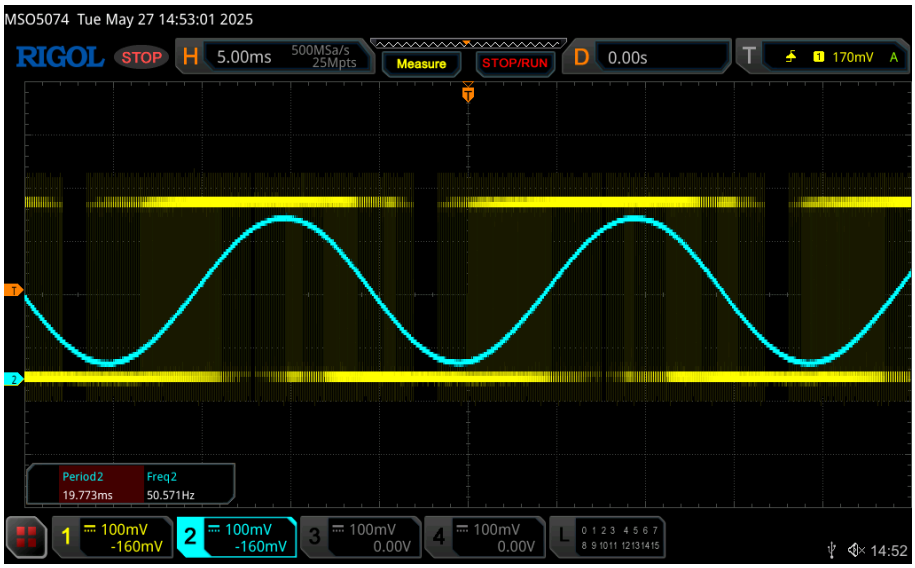


Figure 8: Sine wave generation - sine period and frequency

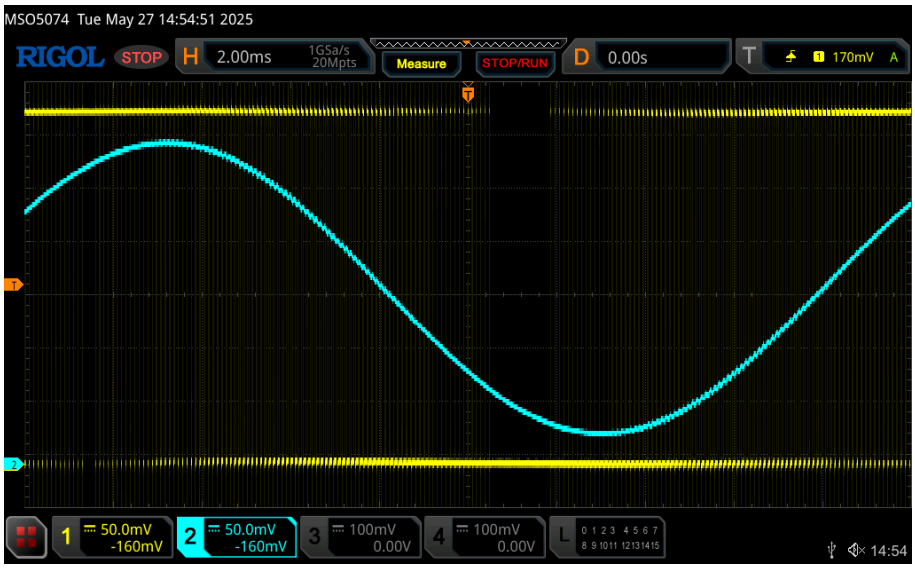


Figure 9: Sine wave generation - detailed view of the signals

3 Conclusion

This assignment has been essential in gaining hands-on experience with the STM32 HAL (Hardware Abstraction Layer) libraries, as opposed to relying on the lower-level (LL) drivers. By focusing on PWM signal generation, the lab provided practical insights into configuring timers, adjusting duty cycles, and applying PWM in various use cases, such as LED dimming or wave generation. These exercises not only reinforced theoretical knowledge but also highlighted the advantages of using HAL for faster development and improved code portability across STM32 microcontrollers.

4 Sources and notes

All the projects in this document have been developed from scratch using STM32CubeIDE software, with reference to the STM32 MCU's Reference Manual and User Manual, as well as the Nucleo board's User Manual—all provided by STMicroelectronics. These specific resources can be found on `https://www.st.com/` by searching for the Nucleo-F401RE board and navigating to the documentation section. They are also available within the Digital System Electronics course material.

The members of the group have used GitHub to collaborate in the developing of the delivered codes. For each project, only the essential files have been provided. However, the project does not always work with just these files alone. For exercises requiring the STM32Cube hardware configuration, it is necessary to start a new project in STM32CubeIDE using the delivered .ioc file. After generating the C code, the `main.c` (and other files, if necessary) must be replaced with the delivered version. Only after completing these steps will the project be ready to be built.

Since the time available in laboratories was not enough, only for the testing phase of the project of Section 2.3, a different oscilloscope has been used. It is the model MSO5074, manufactured by Rigol (more technical details at this link). The probes were PVP3150, manufactured by Rigol.