***Introduction to Robotic Systems Course***

**LAB 03a**

**Analog Output with PWM**

Contents

[1 Introduction 1](#_Toc30053024)

[1.1 Lab Overview 1](#_Toc30053025)

[2 Requirements 1](#_Toc30053026)

[3 Pulse-width Modulation 1](#_Toc30053027)

[4 Timer in STM32 Devices 1](#_Toc30053028)

[4.1 Systick Timer 1](#_Toc30053029)

[4.2 General-purpose Timers 2](#_Toc30053030)

[5 Task: Download and Open Project Folder 2](#_Toc30053031)

[5.1 Mapping Output to PWM-enabled Pins 2](#_Toc30053032)

[6 Task: Edit Main.c File 2](#_Toc30053033)

[6.1 Include Header Files 2](#_Toc30053034)

[6.2 Declare and Initialize Variables 3](#_Toc30053035)

[6.3 Interrupt Service Routine 3](#_Toc30053036)

[6.4 Main Function 4](#_Toc30053037)

[7 Test Code on the Board 5](#_Toc30053038)

# Introduction

## Lab Overview

In this lab, you will program the microcontroller to output a pulse-width modulated signal (PWM) that will be used to control the voltage supplied to a LED, thus controlling its brightness. The program will be written in C language.

# Requirements

The following hardware and software are required to complete this lab:

* **Hardware:** 
  + OpenCR1.0 Microcontroller Board and ULINK-ME debugger
  + LED, 270Ω Resistor, Breadboard, and jumper cables
* **Software:** Keil MDK. Version v5.28 (course was developed with this version).

# Pulse-width Modulation

A PWM signal refers to a signal in which the ratio of the pulse width to the pulse period is controlled using a modulation technique or algorithm. PWM allows for digital control that otherwise would have required an analogue signal. The output level of a PWM signal can be calculated as the average of the pulses.

# Timer in STM32 Devices

Embedded in most STM32 devices is the Systick Timer (SYSTICK) and a general-purpose timer. Our C program will control the operations and features of these timers that will be combined to generate a PWM signal.

## Systick Timer

The Systick Timer is a system timer that can be configured to provide a periodic interrupt signal to the processor, which when serviced will cause the processor to update the PWM output.

## General-purpose Timers

The PWM pulse width and pulse period will be determined by the general-purpose timer. The general-purpose timer is used to provide timing resources for hardware and software tasks.

# Task: Download and Open Project Folder

Download and open the Keil MDK project provided for this lab. For most of the lab, you will need to edit the **main.c** file.

## Mapping Output to PWM-enabled Pins

In the Keil project provided for this lab, we have mapped the dedicated PWM pins for you in the **platform.h** file.

We can note the following from the OpenCR1.0 manual:

* Pin No 6 on the Arduino connector can be used to output a PWM signal using channel 3 of Timer 2 when alternative function 01 is selected. See Section 3.1 (Arduino Connector) of the manual.
* GPIOA, GPIO\_PIN\_2 corresponds to Pin No 6 of the Arduino Connector. See Section 3.9 (Pin Definition) of the manual. Therefore, this pin will be configured to operate in PWM mode.

On the opened MDK project, open the **platform.h** file; this PWM pin is PA2 that is defined as shown in the line below.

PA2 = (0 << 16) | 2,

There are other Timer pins in the manual, can you identify them?

# Task: Edit Main.c File

Open the **main.c** file in your MDK project. The **main.c** file contains two functions:

* The interrupt service routine (ISR) called **systick\_callback\_isr**.
* The main function.

## Include Header Files

At the top of the **main.c** file, add the following lines of codes (shown in the code block below) to include the required header files. These header files contain the platform-specific ports definitions and timer functions. These header files can also be seen in the MDK project, you can open them and take some time to study them.

1. #include <platform.h>
2. #include <timer.h>
3. #include <gpio.h>

## Declare and Initialize Variables

In this step, you will declare and initialize the PWM frequency, duty cycle, and counter direction variables. We will also declare the timer period and PWM period variables at this stage. Immediately below the include header statements, enter the following lines of code:

1. static double PWM\_frequency = 0.001;
2. volatile double dutycycle = 0.0;
3. uint32\_t timer\_period;
4. uint32\_t trigger\_point;
5. int dutycycle\_change\_direction = 0;

The variable “PWM\_frequency” is used to initialize the PWM frequency in KHz, which is 0.001 MHz of the system clock.

The variable “dutycycle” is declared volatile because its value will change as the duty cycle is updated.

## Interrupt Service Routine

Next, edit the **systick\_callback\_isr** function. The **systick\_callback\_isr** function will be called when an interrupt is received from the systick timer.

Inside the **systick\_callback\_isr** function, add the following line of code:

1. if (dutycycle < 1 && !dutycycle\_change\_direction)
2. {
3. dutycycle += 0.05;
4. }
5. else if (dutycycle > 0 && dutycycle\_change\_direction)
6. {
7. dutycycle -= 0.05;
8. }
9. else
10. {
11. dutycycle\_change\_direction = !dutycycle\_change\_direction ;
12. }

From the above code, the duty cycle is incremented or decremented by 0.05. The choice whether to increment or decrement is determined by the value of the duty cycle and the value of the **dutycycle\_change\_direction** (0 or 1). The **dutycycle\_change\_direction** is toggled when the duty cycle is 1 or 0.

Inside the **systick\_callback\_isr** function, and outside the if else statement, add the following lines:

1. trigger\_point = (uint32\_t) (timer\_period\*dutycycle) - 1;
2. TIM2->CCR3 = trigger\_point;

In the two lines of code above, the first line updates the PWM period every time the duty cycle changes and the second line updates the “timer capture and compare” register for channel 3 of Timer 2.

## Main Function

Inside the main function, add the following code:

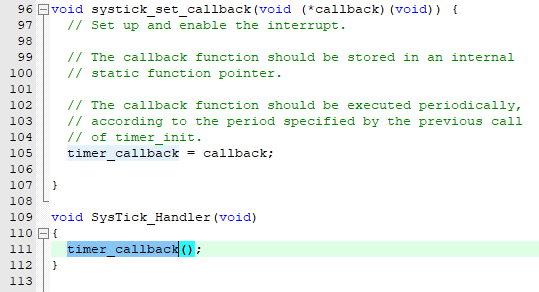
1. timer\_period = (SystemCoreClock / 1e6) / PWM\_frequency - 1;
2. gpio\_set\_mode(P\_PWM, AF);
3. gpio\_AF\_config(P\_PWM, GPIO\_AF\_TIM2);
4. timer\_init(timer\_period);
5. trigger\_point = (uint32\_t)(timer\_period\*dutycycle) - 1;
6. PWM\_init(trigger\_point);
7. systick\_init (50);
8. systick\_set\_callback(systick\_callback\_isr);
9. timer\_enable();

See Table 1 for the description for each line of the above code.

*Table 1: Description of each line of code inside main function*

|  |  |
| --- | --- |
| **Line number** | **Description** |
| 1 | The timer period is set in KHz. It is derived from the system clock. |
| 2 | The pin **P\_PWM** is set to perform an alternative function. **P\_PWM** is defined as PA2 in the **platform.h** file. |
| 3 | The pin **P\_PWM** alternative function is configured to operate on channel 3 of Timer 2. **GPIO\_AF\_TIM2** is defined in the **platform.h** file. |
| 4 | The timer is initialized to the timer period. |
| 5 | The PWM period is initialized by computing the initial trigger point. |
| 6 | This line sets the timer to enable PWM output and sets the trigger point of the timer used. **PWM\_init** is defined in the **timer.c** file. |
| 7 | This line sets the systick interrupt for every 50 ms. |
| 8 | The interrupt service routine function is defined. In the next paragraph, this is explained. |
| 9 | The timer is enabled. |

In Line 8, the interrupt service routine function is called. It has an input argument, which is the earlier defined **systick\_callback\_isr function**. To understand what is going on, open the **timer.c** file, which is part of the opened project files, and examine these two functions shown in Figure 1.



*Figure 1: Code snippet of interrupt call back function.*

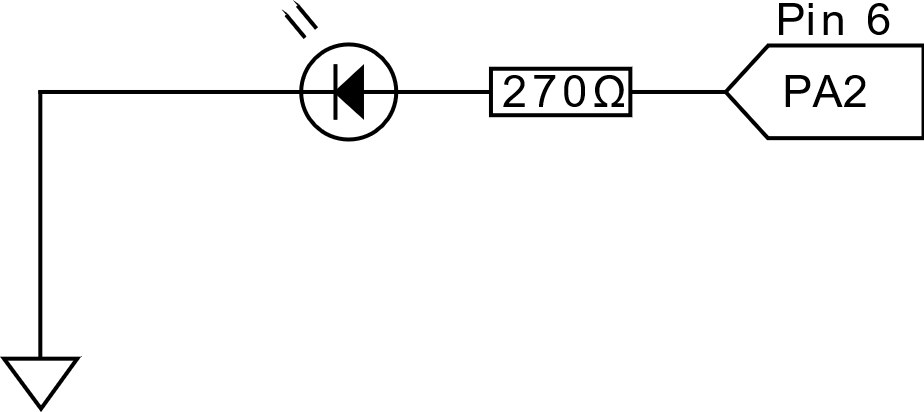
**SysTick\_Handler** is called when a SysTick interrupt is received. The **Systick\_Handler** function calls **timer\_callback()** that is set by the **systick\_set\_callback** function.

Finally, add an infinite while loop that contains the instruction **\_\_WFI();** as shown in the code below. This instruction causes the processor to go into a low-power standby state that is only exited when an interrupt occurs.

1. while (1) {
2. \_\_WFI();
3. }

# Test Code on the Board

Ensure that the LED circuit is connected as shown in Figure 2. Connect the OpenCR1.0 board to the computer through the debugger.



*Figure 2: Schematic of the LED circuit connected to the OpenCR1.0 board.*

Compile the code and then download the code to the flash memory. You will notice the LED brightness varying periodically.