Experiment 7: Pulse-Width Modulation

1.0 Introduction

A microcontroller's interaction with the real world (and real humans) often requires the generation of voltages that can vary over a continuous spectrum. Although your microcontroller has an on-board digital-to-analog converter (DAC), it is too weak to drive any load that requires high power. Even with amplification, a DAC is inefficient for the purpose of driving heavy loads. Digital electronic systems are well-suited to switching circuits on and off. In circumstances where a load can be switched on and off in such a way that the average output voltage is treated (or perceived) as the signal, a solution is Pulse-Width Modulation (PWM). In this experiment, you will learn about configuring and using PWM to control a high-power load (a light) and generate a variable duty-cycle waveform.

Step		Points
4.0	Prelab	10
6.1	Wiring	10
6.2	Simple PWM	10
6.3	Debounce keypad	10
6.4	RGB color mixing control using keypad	10
7.0	Post lab submission	*
Total		**50**

***All lab points are contingent on submission of code completed in lab.**

2.0 Objectives

1. To understand the concepts of pulse-width modulation

3.0 Equipment and Software

1. Standard ECE362 Software Toolchain (Installation instructions available in Experiment 0: Getting Started)
2. STM32F0-DISCOVERY development board (For procurement instructions, see Experiment 0: Getting Started)

4.0 Prelab

Read over the experiment background in section 5, wire you development system as shown in Figure 3, and complete the prelab assignment on the course web page.

5.0 Background

5.1 Pulse-Width Modulation

Consider a digital logic circuit which outputs a repeating, periodic signal over a known time period, t_S. Within the signal period, the circuit can output logic '1' for a subinterval, t_H, and for the remaining time, outputs a logic '0'. This is illustrated in figure 1, below

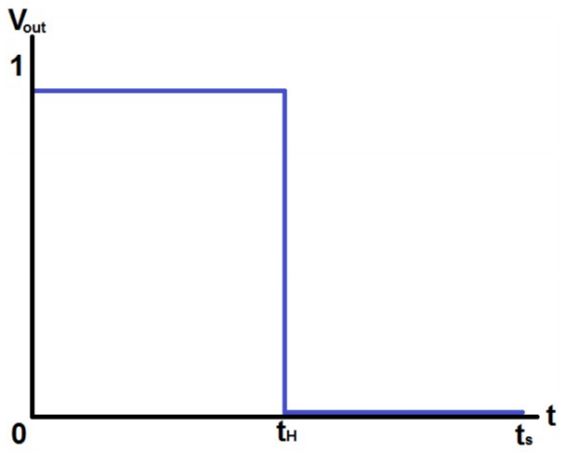

Figure 1. Simple Periodic Output

Expanding on the output above, suppose the circuit can be made to vary the logic '1' interval t_H to any value between 0 and t_S. A variety of potential outputs could result:

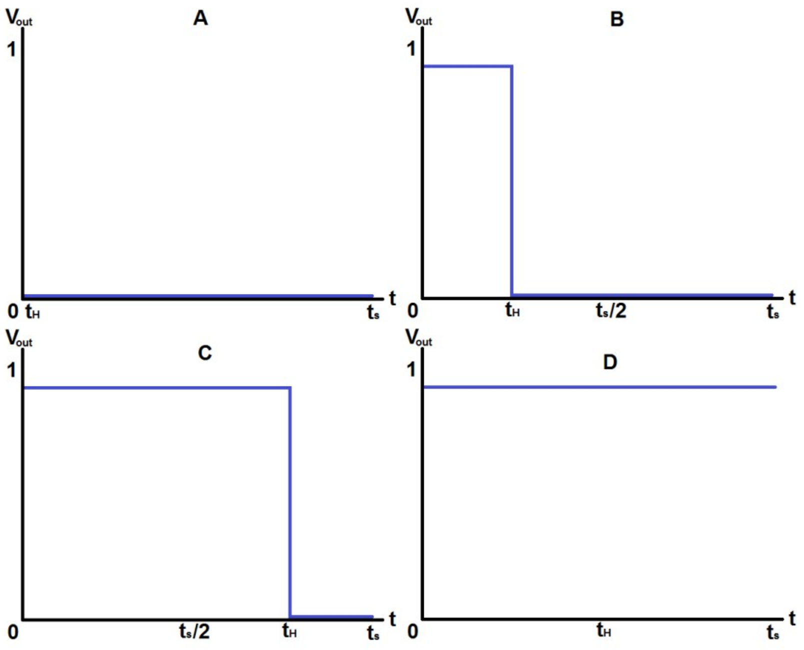

Figure 2. Periodic Output a) t_H = 0 b) $t_H = t_S/4$ c) $t_H = 3t_S/4$ d) $t_H = t_S$

The relationship between the time the periodic signal spends high (tH) and the total period of the repeating signal (ts) is known as the signal's duty cycle and is usually expressed as a percentage. In figure 2, signal 2a could be said to have a 0% duty cycle, signal 2b, 25%, 2c, 75%, and 2d, 100%. By controlling the duty cycle of the repeating, periodic output signal, a pulse-width modulator is capable of controlling average delivered current or power to a given load, and can further be used, possibly in conjunction with filtering, for crude approximations of analog values. Sometimes these values must be low-pass filtered to yield an analog value. For devices such as LEDs, the human visual system acts as a natural low-pass filter.

5.2 PWM on the STM32F0

On STM32 microcontrollers, pulse-width modulation is integrated into the various TIM peripherals. For the purposes of this experiment, TIM1 will be used for PWM outputs; you may wish to revisit the basics of timer operation from Lab Experiment 5.

As with all other peripheral configurations, the first step for configuring a peripheral is to enable the clock signal to it. This is done within the reset and clock control (RCC) system of the processor. Use of PWM, as well as other non-analog microcontroller peripherals, requires the use of alternative pin input functions. Alternate pin functions are detailed in the pin definitions table of the STM32F051R8T6 datasheet. Alternate function mode is specified using the GPIOx_MODER register for the appropriate GPIO port. Some pins can have more than one alternate function associated with them; specifying which alternate function is to be used is the purpose of the GPIOx_AFRL and GPIOx_AFRH registers.

Each timer peripheral on the STM32F0 features 4 independent channels, each of which can be set up to output a PWM signal. Specifying a channel as a PWM output requires writing to the TIMx_CCMRx registers. In PWM mode, the PWM frequency is specified through the TIMx_ARR register and duty cycles of the channels are controlled through the TIMx_CCRx registers. Aside from specifying alternate function mode on the associated I/O, additional steps must be taken in order to route the PWM signal from the internal peripheral to the external I/O pin. PWM is part of the timer's capture/compare subsystem, thus the capture compare output for the selected channels must be activated in the timer capture/compare enable register, TIMx_CCER. The TIM1 subsystem is very similar to the TIM3 subsystem which was studied at length in lecture. One difference with TIM1 is that it has a "break and dead-time register" (BDTR). We will not use these features for this experiment, but **the MOE bit of this register must be enabled to generate output on any of the channels of TIM1**. At the end of timer configurations, remember to enable the timer, in the TIMx_CR1 register. For further assistance in setting up PWM, consult the example in appendix A.9.8 of the family reference manual.

5.3 Debouncing a keypad

Because switches are mechanical devices, they bounce. This means we do not get a stable electrical signal on a press or a release. Because of switch bounce, the voltage fluctuates between Vdd and ground before settling down to a stable value. There are several ways to fix this. Debouncing can be performed in hardware or software.

1) Hardware debouncing: You have already witnessed this in previous labs (lab 3). Hardware debouncing is done via a simple RC network to filter high frequency glitches that occur from mechanical bounce.

2) Software debouncing: In certain applications it might be beneficial to debounce switches in software rather than use an RC filter on each switch.

Software debouncing can be achieved in a variety of methods. The simplest debouncing method is to read the voltage, wait for the switch to settle and read it again. However there are more robust approaches, which are more suited for a keypad matrix. Here we describe one such method that we dub the "history method".

In this “history” method we store N previous button states. Each button's history is represented by a single variable. This variable is referred from here on, as the 'history variable'. The least significant bit of the *history variable* represents the most recent button state and the most significant bit represents the oldest button state in history.

When a new button state is read, the *history variable* is shifted left by one, making space for the new state in the least significant bit. The new state value (i.e. 0 or 1) is ORed into the *history variable*.

Now that we have a history of button states we can look for two specific patterns to identify a button press and release. A 0b0000 0001 represents a button press and 0b1111 1110 represents a button release. Note these patterns are applicable when the button's logic is active high, else the patterns are switched. Therefore, by checking if the history variable equals 1 (i.e. 0b0000 0001) we can check for a button press. Similarly by checking if history variable is -2 (i.e. 0b1111 1110) we can check for a button release.

For this experiment we will implement the 'history' method of debouncing the keypad. For each key on the keypad, an 8-bit unsigned variable is used. The 16-key history is stored as an array in RAM. The array's first 4 elements represent the history for the first column of keys, the elements from 4-7 represent the history values of the second column of keys and so on.

6.0 Experiment

In this lab experiment will use PWM to control the RGB LED to perform color mixing. In addition, you will debounce the keypad and use it as input to control the PWM frequency and duty cycle of the three LEDs. You are also given an object file, **lcd_driver.o**, that interfaces with the LCD screen which you will use to output PWM information. Install the lcd_driver.o file in the src/ folder of your System Workbench project and add it to your project as directed by the instructions on the course web page. (https://engineering.purdue.edu/ece362/refs/eclipse/pre-compiled-module.html)

6.1 Wiring

Connect the RGB LED, the keypad, and the LCD screen to the STM32 discovery board as shown in Figure 3. First, you’ll need to connect the RGB LED to the STM32. The RGB LED is a common anode configuration. Connect the anode to 3V and the three RGB leads to TIM1_CH3, TIM1_CH2 and TIM1_CH1, respectively. You should be able to place the LED directly between the 3V power rail and the pins on the STM32 without using additional wires.

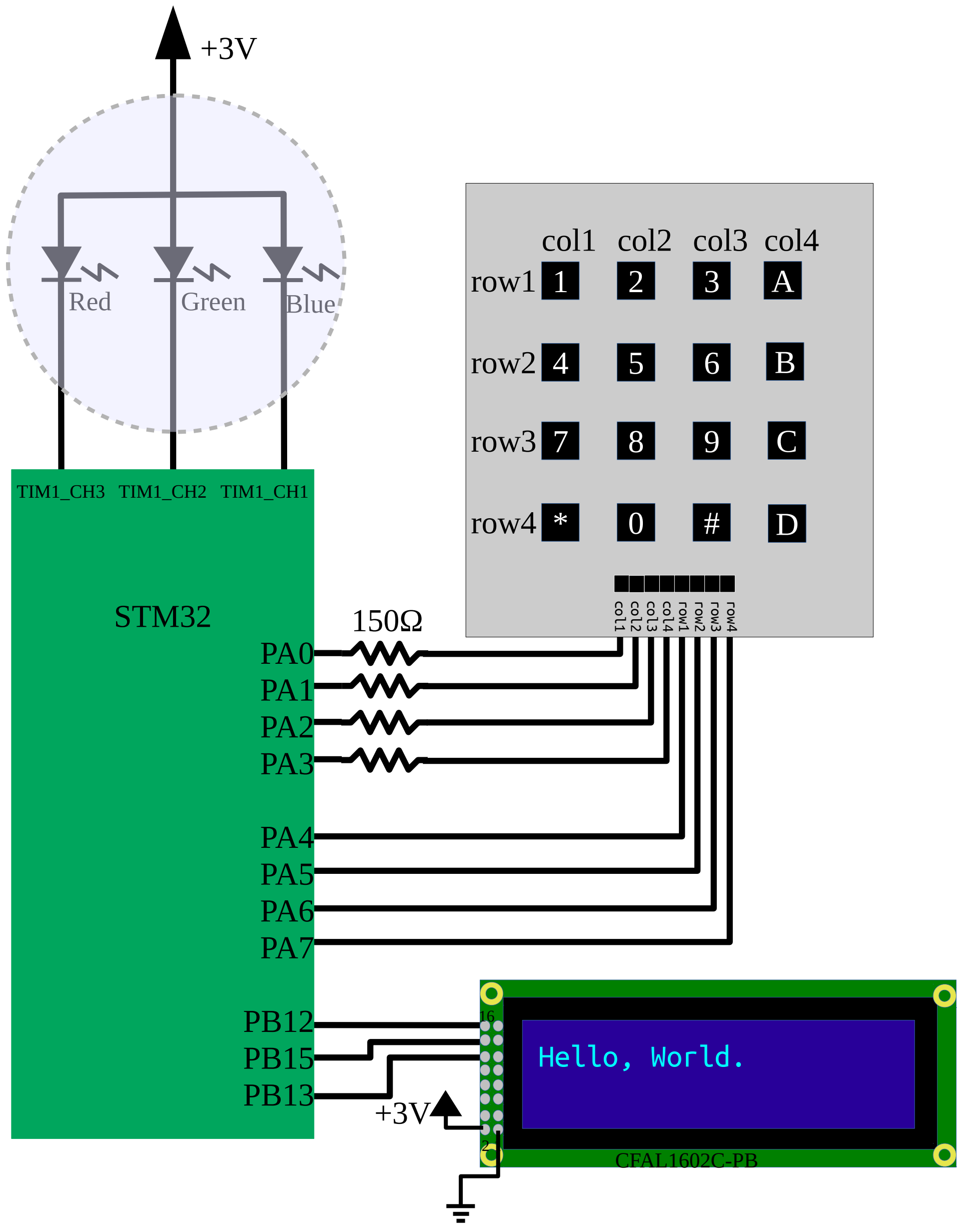

Figure 3. Wiring for lab experiment 7.

When you have connected all components for the lab experiment, uncomment **`test_wiring()`** in the main.c file. You will observe the following:

1) When you press any key it is displayed on the second line of the LCD
2) The text, “This is a test” appears on the first line of the LCD
3) The RGB LED lights up

Demonstrate this to your TA.

6.2 Simple PWM

This section, you will need to complete four functions:

1) **`setup_gpio()`**: Should enable the clock to GPIO port A, and configure the modes of the three pins corresponding to TIM1_CH1, TIM1_CH2 and TIM1_CH3 with alternate functions.
2) **`setup_pwm()`**: Should configure TIM1 so that it is non-centered and upcounting. All channels should be set for the PWM1 output compare mode. The prescaler output should be 100 kHz, and choose the value of ARR so that the PWM frequency is 1 kHz. The duty cycle of each channel will be set by writing a value between 0 and 100 to the CCRx registers. **Note that since we have a common anode configuration, CCRx of 100 will result in a LED being off and a CCRx of 0 will result in maximum brightness.**
3) **`update_freq()`**: This function accepts an integer argument that is used to update the TIM1_PSC to produce the requested frequency (as close as possible) on the output pins. Remember that the output frequency will be 100 times slower than the TIM1_PSC due to TIM1_ARR always being 100-1 (99). The formula for calculating the output frequency is then:

   freq = 48,000,000.0 / (TIM1_PSC + 1) / 100.0

   You should determine the formula to use to put the proper value into TIM1_PSC given the frequency. Use floating point values to do the calculation to avoid rounding errors. Although TIM1_PSC is an integer, there is no problem with a C program assigning a floating point result to it. It is automatically rounded and converted to an integer.
4) **`update_rgb()`**: This function accepts three arguments – the red, green, and blue values used to set the CCRx registers. The values should never be smaller than zero or larger than 100. The value can be assigned directly to the appropriate CCR registers. E.g., the red LED is connected to channel 1.

Uncomment **`prob2()`** in your main program. You should see red fade to green which fades into blue.

Demonstrate this to your TA.

6.3 Debounce keypad

For this section you will debounce the entire keypad. To understand the logic of debouncing, read though the background section of the lab. The keypad is scanned by driving the columns and observing the rows of the keypad. You need to complete the following functions:

1) **`init_keypad()`**: This function should enable the clock to port A, configure pins 0, 1, 2 and 3 as outputs (we will use these to drive the columns of the keypad). Configure pins 4, 5, 6 and 7 to be inputs with a pull-down resistor (these four pins connected to the rows will being scanned to determine the row of a button press). If you do not configure a pull-down resistor, the voltage level will float high or low and you will produce very strange results.
2) **`setup_timer6()`**:

   This function should do the following:

   - Enable clock to Timer 6.
   - Set PSC and ARR values so that the timer update event occurs exactly once every 1ms.
   - Enable UIE in the TIMER6's DIER register.
   - Enable TIM6 interrupt in NVIC's ISER register.
3) **`get_key_press()`**: The functionality of this subroutine is described below, type a syntactically correct C code in your main.c

```
int get_key_press() {
  while(1) {
    for(int i = 0; i < 16; i++) {
      // Check if history[i] == 1
      // if so return i
    }
  }
}
```
4) **`get_key_release()`**: The functionality of this subroutine is described below, type a syntactically correct C code in your main.c

```
int get_key_release() {
  while(1) {
    for(int i = 0; i < 16; i++) {
      // Check if history[i] == -2
      // if so return i
    }
  }
}
```

5) **`TIM6_DAC_IRQHandler()`**: The functionality of this subroutine is described below, type a syntactically correct C code in your main.c
   a. Acknowledge TIM6 interrupt
   b. set a local variable "row" equal to (GPIOA->IDR >> 4) & 0xf;
   c. set a local variable "index" equal to col << 2
   d. left shift "history[index]" by 1 and store it back
   e. AND "row" & 0x1 and OR it into "history[index]"
   f. Now repeat (d) for index+1, index+2 and index+3
   g. Repeat (e) for index+1, index+2 and index+3, the only difference being, OR "(row >> 1) & 0x1" into history[index+1] instead of "row", similarly OR "(row >> 2) & 0x1" for history[index+2] and so on.
   h. Increment col by 1
   i. if col > 3, set col to 0
   j. Set GPIOA->ODR = (1 << col);

Uncomment **`prob3()`**. When you press any key, the LCD will display which key was pressed.

Demonstrate this to your TA.

6.4 RGB color mixing control using keypad

For this section you will need to complete the function **`prob4()`**.

You should make the following assumptions:

1) **`get_key_pressed()`** returns the current key that is pressed, returns 0xf if '*' and 0xe when '#' is pressed.
2) **`get_user_freq()`** returns the frequency entered by the user. It will wait for 6 digits to be pressed, or the number can be terminated by pressing 'D'.
3) **`get_pwm_duty()`** updates the duty cycles in variables 'red', 'grn' and 'blue'. Enter two digits for each color. E.g., to turn red on fully, green on at half intensity, and blue on at quarter intensity, enter 995025. You must enter 6 digits.

In the while(1) loop in **`prob4()`** add logic to does the following:

a) Check if '*' is pressed. If it is, call **`get_user_freq()`** and **`update_freq()`** to update the PWM frequency.
b) Check if '#' is pressed. If it is, call the **`get_pwm_duty()`** and **`update_rgb()`** to update the PWM duty cycles for 'red', 'grn' and 'blue'.

Uncomment **`prob4()`**. Once completed, you should be able to control PWM frequency and duty cycle using the keypad.

Use the oscilloscope at your station to view the voltage level on the red LED pin. Remember that you should never try to attach scope leads directly to pins on your STM32 development board. Instead, you should attach the lead to a ground wire and the red lead on the RGB LED. You should be able to demonstrate any frequency and duty cycle that your TA suggests.

Demonstrate this to your TA.

7.0 Complete post lab and submit code

Submit your code and complete the post lab on the website. The portal for post-lab will close 10 minutes after your scheduled lab.

8.0 Clean up your lab station and log out

Demonstrate to your TA that your lab station is clean and that you have logged out of your workstation