# ECE 362 Lab Experiment 5: Timers

## 1.0 Introduction

The timer subsystem allows the microcontroller to produce waveforms, and respond to input without using the CPU. In past experiments, we wrote programs that set GPIO pins on and off to strobe LEDs and read push buttons. In this experiment, we will set up timers to do things like this without direct CPU involvement. Although a program is used to configure the control registers, it does not need to continually coordinate with the hardware to produce actions. Because the timer subsystem has so many control registers, we will use C structure and field constructs to do the timer configuration. Wherever possible, use the CMSIS symbol designations when you write your code.

| STEP | DESCRIPTION | POINTS |
|---|---|---|
| 4.0 | Prelab Questions | 20 |
| 6.1 | Turning on the green LED with C | 5 |
| 6.2 | A timer-based LED flasher | 5 |
| 6.3 | A two-LED flasher | 5 |
| 6.4 | Flashing LEDs out of sync | 5 |
| 6.5 | Wire a 7-segment LED display | 10 |
| 6.6 | A bouncy pushbutton reader | 10 |
| 6.7 | A debounced pushbutton reader | 10 |
| 6.8 | Simultaneous timers | 10 |
| 7 | Post-lab questions and code submission | all of them* |
| | **TOTAL** | 80* |

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

## 2.0 Objectives

1. To practice writing register configuration code in C
2. To configure the timer subsystem to produce periodic outputs
3. To configure the timer subsystem respond to input
4. To reliably debounce a pushbutton switch

## 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 section 5 “Background,” and complete the prelab on the course website.

## 5.0 Background

### 5.1 Structures and Fields

We learned that groups of control registers are defined as a base address in memory with individual registers accessed at an offset from that base address. In C, a structure is just a contiguous chunk of memory and the fields represent offsets into that memory. These two concepts are similar because C was designed as a language to work directly with hardware. We have typically used constructs like this to access and modify control registers:

```
.equ RCC, 0x40021000
.equ AHBENR, 0x14
.equ GPIOCEN, 0x80000
.global enablePortC
enablePortC:
      push {lr}
      ldr r3, =RCC
      ldr r0, [r3, #AHBENR]
      ldr r1, =GPIOCEN
      orrs r1, r0
      str r1, [r3, #AHBENR]
      pop {pc}
```

In C, we can do the same thing using structure and field notation, where the *pointers to structures* as well as field offsets are defined in standard header files:

```
#include “stm32f0xx.h”
#include “stm32f0308_discovery.h”

void enablePortC(void) {
      int tmp = (*RCC).AHBENR;
      tmp = tmp | RCC_AHBENR_GPIOCEN;
      (*RCC).AHBENR = tmp;
}
```

We can use arrow operator for access to a field of a *pointer to a structure* to condense the program.

```
#include “stm32f0xx.h”
#include “stm32f0308_discovery.h”

void enablePortC(void) {
      int tmp = RCC->AHBENR;
      tmp = tmp | RCC_AHBENR_GPIOCEN;
      RCC->AHBENR = tmp;
}
```

Finally, we can use the shorthand for OR assignment to eliminate the need for a temporary variable.

```
#include “stm32f0xx.h”
#include “stm32f0308_discovery.h”

void enablePortC(void) {
      RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
}
```

### 5.2 Interrupts in C

In experiment 4, we created interrupt service routines (ISRs) in assembly language to handle exceptions like the SysTick and EXTI0_1_IRQ interrupts. All that was necessary to create an ISR was to write a subroutine in assembly language, make it .global to override the .weak symbol defined by the firmware, and add a .type designation to tell the linker that the label was actually a function rather than something like data. (Because the vector table entry must have the LSB set to indicate to the CPU that it is a Thumb subroutine.)

```
.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
      push {lr}
      ...
      pop {pc}
```

C is the same way, except that the C compiler automatically designates subroutines as .global, sets up the .type as a function, and pushes and pops the right registers, so the same ISR in C is:

```
void SysTick_Handler(void) {
...
}
```

## 6.0 Experiment

For this experiment, select the Standard Peripheral (StdPeriph) firmware for your project. For your project, you are provided two files: main.c and support.s. In the support.s file, you will find the **`nano_wait`** and **`micro_wait`** assembly language subroutines. In main.c, your **`main()`** function should call one of several C functions that you write. Uncomment one at a time to demonstrate your code to your TAs. You may choose to demonstrate several of the functions at once, one after the next.

### 6.1 Turning on the green LED with C

Write a C function named **`prob1()`** that sets up your development board to repeatedly turn on the green LED, pause for one second using either **`nano_wait`** or **`micro_wait`**, turn it the green LED off, and pause again for one second. The green LED (LD4) is connected to Port C, pin 9. You should take care that your configuration code does not change the mode configurations for any other pins. You may use the examples you have seen in lecture for turning on the blue LED and modify them appropriately. <u>Demonstrate this functionality to your TA</u>.

## 6.2 A timer-based LED flasher

Create a function, **`prob2()`**, that sets up the appropriate timer and channel to repeatedly toggle the green LED at a rate of one second on and one second off. Use the example you have seen in lecture as a basis for your code. Adapt the code to configure the timer and channel for the green LED instead of the blue LED.

At the end of the function, instead of occupying the CPU with an empty loop ( **`for(;;);`** ), use the following loop with an inline assembly statement:

```
while(1)
      asm(“wfi”);
```

This code will execute the “wait for interrupt” instruction which stops the CPU from executing instructions until it receives an interrupt. (By the way, the debugger interface triggers interrupts, so it will wake up the CPU each time you interrupt your program.) Demonstrate this functionality to your TA.

## 6.3 A two-LED flasher

Create a function, **`prob3()`**, that sets up the appropriate timer and channels to repeatedly toggle the green LED as in section 6.2. This time, flash the Blue LED (LD3) on PC8 at the same time and same rate as the green LED. Demonstrate this functionality to your TA.

## 6.4 Flashing LEDs out of sync

Create a subroutine, **`prob4()`**, to flash the LEDs attached to PC8 and PC9 at the same rate as in step 6.3, but this time use different values for the CCRx registers to toggle each LED at a different time. Make the sequence spaced evenly. E.g. The LEDs should follow an evenly-spaced and regular on-on-off-off pattern. Specifically, the blue LED should turn on first. 0.5 seconds later, the green LED should turn on. 0.5 seconds after that, the blue LED should turn off. 0.5 seconds after that, the green LED should turn off. You should set the timer CNT register before enabling the timer to make the sequence go in the correct order. Demonstrate this functionality to your TA. Your TA will ask you to change the CNT register to make the sequence go in the opposite order.

### 6.5 Wiring a 7-segment LED display

Consult the documentation for the 7-segment LED on the ECE 362 reference page. Unplug your STM32 from its power source, and wire a 7-segment LED from your lab kit to your STM32 so that the following pins are connected:

| STM32 | LED |
|---|---|
| PC0: | A |
| PC1: | B |
| PC2: | C |
| PC3: | D |
| PC4: | E |
| PC5: | F |
| PC6: | G |
| PC7: | DP |

Also wire the LED common anode to **3v**.

Each segment of the 7-segment LED is rated to handle 25mA at most, and the STM32 will source 25mA per GPIO pin, so this circuit can be completed safely. If you are concerned about your wiring, let your TA check it before you power it on.

Write a function called **`init_display()`** that enables port C and configures only pins PC0 – PC7 to be outputs and also sets bits 0 – 7 of the output to be '1'. Also write a function called **`display()`** that accepts a single int argument that describes a hexadecimal digit to display. Since the LED is a common anode configuration, each segment will be lit by a low voltage on the output of the corresponding STM32 pin. To illuminate the digit '0', the following segments should be lit: A, B, C, D, E, F. That means that PC0 – PC5 should be set low, and PC6 should be set high. A portion of the **`display()`** function has been written for you. It contains a variable named **output** which is an array of characters (8-bit integers) that define the patterns to use for digits. The output value for the '0' and '1' digits have already been written for you. Write the patters for the other hexadecimal digits from '2' to 'f' using the descriptions provided. You can look up the pattern for the digit **x**, which was passed as a parameter. Complete the **`display()`** function by using the 8-bit value from the **output** table to set the output bits of PC0 – PC6 (but not PC7).

Uncomment the **`test_display()`** function in your **`main()`** function to run a test that displays each of the hexadecimal digits '0' – 'f' for 0.5 seconds. If you are able to complete this step with no wiring errors or encoding errors in your output table, you are surely super-human. Nevertheless, you may claim that you did when you demonstrate this step to your TA or wait until the next step when you can show the display working correctly.

### 6.6 A bouncy pushbutton reader

Create a global variable named **count** and C function named **increment** that do the following:

```
int count = 0;
void increment(void) {
      count += 1;
      display(count);
}
```

Create an interrupt service routine for Timer 2 named **TIM2_IRQHandler()**. Remember that you must name it correctly (with the same spelling and case) for it to work. The interrupt handler should do the following operations:

- Call **increment()**.
- Allocate a local variable.
- Copy the value from **TIM2->CCR1** into the local variable to clear the CC1IF flag.
- Copy the value from **TIM2->CCR2** into the local variable to clear the CC2IF flag.

Next, create a C function **prob6()** that does the following: (An overview of this function is given in lecture.)

- Configure PA0 to be an alternate function as an input to Timer 2.
- Call **init_display()**.
- Call **display(0)**.
- Enable the Timer2 clock in RCC_APB1ENR.
- Set Timer 2's prescaler to 1. (set the PSC register to 0)
- Set Timer 2's auto-reload register to 0xffffffff (all '1' bits).
- Set the direction of Timer 2 channel 1 as "input from Timer Input 1 (TI1)" by setting the CC1S bits to '01' in Timer 2's CCMR1 register.
- Enable Timer 2 channel 1 by setting the CC1E bit in the CCER.
- Enable interrupt generation for Timer 2 channel 1 by setting the CC1IE bit in the Timer 2 DIER register.
- Enable Timer 2 by setting the CEN bit in the Timer 2 CR1 register.
- Enable interrupts with:
  **NVIC -> ISER[0] = 1 << TIM2_IRQn;**
- Terminate the function with a for loop containing the asm("wfi") instruction as before.

When the user pushbutton on your development board is pressed, the output LEDs should show that 1 (if you are lucky) or more interrupts have occurred. When you release the user pushbutton, it may also trigger an interrupt even though you have not configured the timer channel to respond to falling edges.

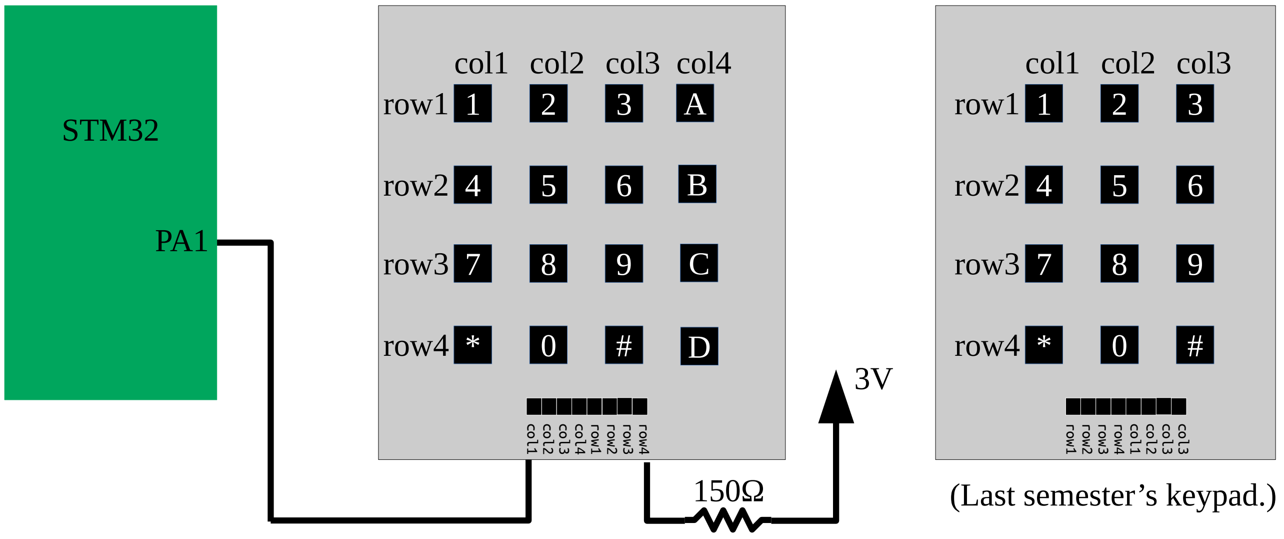
Demonstrate this to your TA.

### 6.7 A debounced pushbutton reader

Create a C function, **`prob7()`**, that does the same configuration as **`prob7()`** except:

- Configure PA1 instead of PA0.
- Configure PA1 to use an internal pull-down resistor using the PUPDR.
- Use Timer 2 channel 2 instead of channel 1.

Add the pushbutton matrix keypad from your development kit to your breadboard. The schematic is as follows:



(Last semester’s keypad.)

A matrix keypad is named as such because its buttons connect a matrix of wires. When the ‘1’ button is pressed, the row1 wire is connected to the col1 wire. (If you are using a 3x4 keypad matrix from a previous semester, the rows and columns are switched, and col3 and col4 are both connected to col3.)

For this experiment, we will use only one button, the ‘*’ key. Use a wire to connect the col1 pin to PA1 of your development board, and connect the row4 pin to 3V via a 150Ω resistor. When the ‘*’ key is pressed, it will raise the PA1 pin high. You will see the same form of “bounce” as you saw in Step 6.4. Now, let us correct it. To do so, add the following configuration steps to **`prob6()`**:

- Turn on input filtering for the IC2F bits of Timer 2’s CCMR. Set the field to all ones (‘1111’) with the following statement:

```
TIM2->CCMR1 |= TIM_CCMR1_IC2F_3 | TIM_CCMR1_IC2F_2 | TIM_CCMR1_IC2F_1 | TIM_CCMR1_IC2F_0;
```

- Set the clock divider to reduce the sample clock by 4 by setting the two CKD bits in Timer 2’s CR1 register to ‘10’.

With these changes, the sample clock is reduced to 2MHz (or 12MHz when you are using the Standard Peripheral firmware). When the ‘*’ is pressed, the input filter further divides the sample clock by 32 to a rate of 62.5kHz (or 375kHz for the Standard Peripheral firmware) and it waits for 8 consecutive positive readings before triggering an interrupt. Each reading is made 16 μsec (2.67 μsec) from the next, so as long as no bounce is detected within the 128 μsec (21.3 μsec) sampling interval, the button press is treated as valid.

When using a 48MHz clock that the Standard Peripheral firmware a sample interval of only 21.3 μsec for the key bounce is fairly short and may not be reliable for some types of buttons. You might try connecting the keypad to pin PA0 to compare how well it works with **`prob6`** of step 6.6. It turns out that the user pushbutton is much worse than the buttons on the keypad. Nevertheless, the keypad bounces are still detected too frequently when no input filtering is used. When input filtering is enabled, the bounces are much rarer.

We know, from lab experiment 3, how to debounce a single button using hardware – even one as bouncy as the user pushbutton. Such hardware is not useful for a matrix of buttons. In later experiments we will revisit debouncing again to reliably read an entire keypad.

Demonstrate your "bounceless" pushbutton to your TA or demonstrate this and the next step together.

## 6.8 Simultaneous Timers

Create an interrupt service routine for Timer 2 named **`TIM3_IRQHandler()`**. Remember that you must name it correctly (with the same spelling and case) for it to work. The interrupt handler should do the following operations:

- Toggle pin PC7 (wired to the decimal point).
- Clear the UIF bit in the Timer 3 SR register.

Next, create a function called **`prob8()`**. It should do the following:

- Enable the clock to Timer3 in the APB1ENR.
- Set the Timer 3 prescaler to divide the system clock by 48000.
- Set the Timer 3 auto reload register so that the counter will reach the value in 0.5 seconds.
- Set the bit in the Timer 3 DIER to enable the update interrupt.
- Enable the timer 3 interrupt in the NVIC.
- Enable Timer 3 to run.
- Call **`prob7()`**.

Notice that you did not set the timer to enable any input or output mode. It's not using any pins, so there's no need for any of this. Each time the counter reaches the ARR value and is updated, it will invoke an "update interrupt." To acknowledge this interrupt, an ISR must clear the UIF bit in the timer's status register. This time, the ISR is not acknowledging the interrupt by reading the CCxIF registers, but by clearing the TIM_SR_UIF bit.

Demonstrate your decimal point-blinking debouncer to your TA.

## 7.0 Post lab submission

Remember to complete the questions (if any) in the postlab submission on the web. You must also submit the entirety of the main.c file you wrote for this lab. **You will receive a zero for this lab if you do not submit your code.**

Postlab submission will close 10 minutes after the completion of your lab section.