ECE 362 Experiment 4: Interrupts

1.0 Introduction

Microprocessors consistently follow a straight sequence of instructions, and you have likely only worked with this kind of programming until now. In this experiment, you will configure a microprocessor to handle exceptional events. These events will be generated in an expected manner, but the principles are the same for failures, faults, and other unexpected events. These events will invoke Exception Handlers that you write. These Exception Handlers will efficiently respond to events only when needed and will not run continually.

Step		Points
4.1	Prelab	15
6.1	Setting up a long-running program	5
6.2	Wiring a seven-segment LED display	5
6.3	Implement a SysTick handler	5
6.4	Implement a GPIO interrupt handler	5
6.5	Reaction timer	10
6.6	Post lab submission	*
Total		**45**

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

2.0 Objectives

1. Wire seven-segment displays to the GPIO pins of a microcontroller using shift registers
2. Practice assembly language techniques
3. Learn about the SysTick and External GPIO interrupts of a microcontroller
4. Implement Interrupt Service Routines

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

4.1 Complete the prelab on the course website

It is highly recommended that you complete the wiring in section 6.2 before lab.

5.0 Background

5.1 Interrupts and Event-Driven Programming

An exception handler (or interrupt service routine) is a hardware-invoked subroutine. There are many types of events that can be configured to generate exceptions. When these events constitute failures, this type of programming is valuable for error handling and fault recovery. More often, we create exception handlers to deal with things we want to occur occasionally, and that we do not want to force the microprocessor to continually check (poll). This is called event-driven programming. Here, the event in question is expected to be handled quickly and only occasionally.

6.0 Experiment

For this lab, we provide two files: main.c and lab4.s. You will fill in the subroutines whose templates are in lab4.s. The lab4.s file also contains many EQU statements to provide the common addresses and offsets you will use to access the control register arrays. Each part of the lab is called by a C subroutine in main.c. As you gradually implement each subroutine in lab4.s, you will build up functionality with assembly language routines. You should ensure that code you complete in lab4.s acts as a proper subroutine, saving any necessary registers, and returning to the caller when complete.

You should also incorporate your completed main.s from lab 3 into the project for this lab to handle the GPIO functions.

6.1 Setting up a long-running program

In this experiment, we model a case where a CPU has something important to work on rather than just sit idly. In section 6.3 we will use fibonacci as the long running program. Copy your fibonacci program from your homework into lab4.s. Test it by uncommenting the **test_fibonacci()** call in main.c. If your program works correctly, it will flash the ~~green~~ blue LED.

Note: test_fibonacci(), will test to make sure that your program is not leaking stack memory (makes sure that you have an equal number of push and pops). To get full credit for this section, your program should not leak memory.

If you are unable to get fibonacci to work, you may take a zero score on this section, and replace the fibonacci call with micro_wait(2500000). You will need to add the micro_wait code into lab4.s.

Show this to your TA.

6.2 Wiring a seven-segment LED display

The wiring for this section uses two shift registers (74HC595). One shift register (LED_DRIVER in the schematic) drives the seven segment displays and the other selects between the two seven segment displays.

Connect the Q_A – Q_G of the shift register to A-G of the seven segment displays and connect Q_H to DPs of the seven segment displays. Use the second shift register's Q_A and Q_C to connect the base of the transistors. You should have a 2N2907 and an SFT1342 from the lab kit you used for ECE 208. If not, you will need to borrow them. You do not need to use one of each. i.e., if you want to use two 2N2907 transistors (with two 150Ω resistors), you may.

We will daisy chain the shift registers so that we can send data to both the shift registers using a three GPIO pins (PC0, PC1 and PC2). So make sure to short (connect) pin 9 of the LED_DRIVER to pin 14 of LED_SELECTOR. The SRCLK in figure 1 is the same as SCK in figure 3. In figure 3 the LED_SELECTORP represents the pins 16 and 8 for the LED_SELECTOR 74HC595 chip. Complete the remaining wiring as shown in the schematic (Figure 3).

Note carefully that the pins for Cathode F and Cathode G of the seven-segment display might not be in the order you wish they were.

You can fit the entire circuit along with the STM32 on a single breadboard, but you will need to plan the placement of your devices carefully. You may want to use an extra breadboard from a previous lab class or from the instrument room.

Make sure the power to your microcontroller is unconnected while you do your wiring. When you are done wiring, double-check that you have not made a mistake (such as connecting 3V to GND). Attach your microcontroller to your development machine, and uncomment only the test_wiring() function in main(). This function is provided to you. It will turn on the two LED segments counting up.

<u>Show your work to your TA.</u>

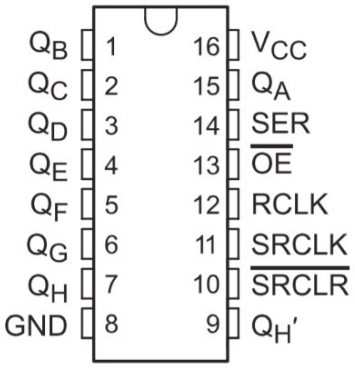

Figure 1. 74HC595 Shift Register

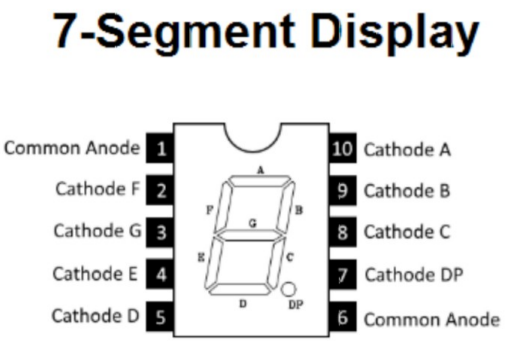

Figure 2. 7-Segment LED Display

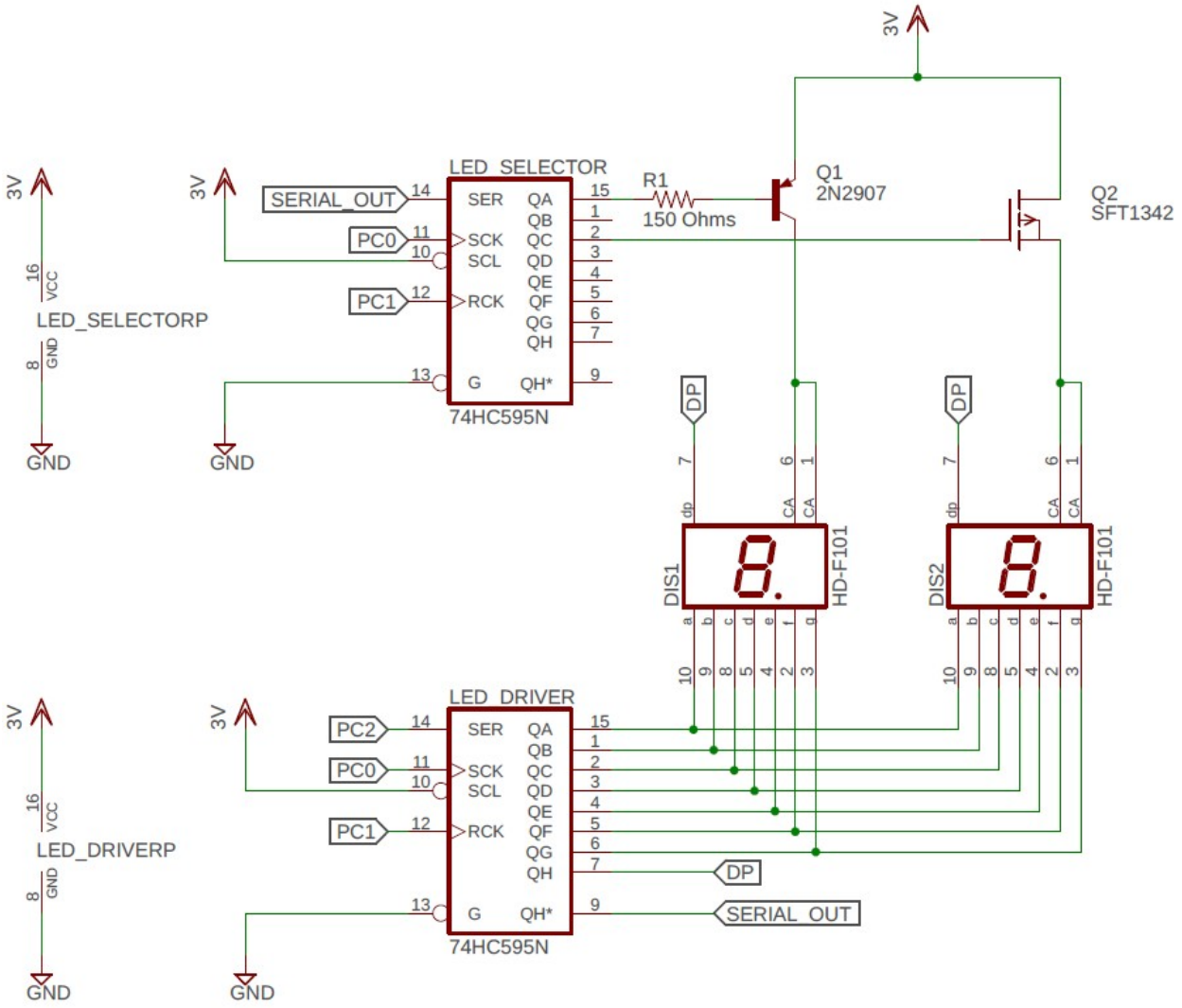

Figure 3. Schematic for wiring

6.3 Implement a SysTick handler

In lab4.s, complete the two subroutines **init_systick** and **SysTick_Handler**. These subroutines will, respectively, initialize the SysTick down counter and reset value, and handle the exception. You should set the counter to use the CPU clock to generate a 100ms second interval repeat (assume a 48MHz CPU clock).

The **SysTick_Handler** subroutine should increment the value of a global variable, **tick_count** in main.c. After incrementing it, it should use the new value as the argument for a call to **display_digit()**.

To test your work, you should uncomment only the **test_SysTick()** call in **main().** It will call your initialization function and then continually run **forever_fibonacci()** which will toggle the decimal point after every calculation of fibonacci(27).

The interrupt will update one seven segment LED, counting up while the forever_fibonacci will toggle the decimal point.

Show your work to your TA.

6.4 Implement a GPIO interrupt handler

This experiment will gradually implement the code needed to configure PA0 (the user pushbutton) as an external interrupt source. Use the structure of code outlined in lecture to build four subroutines:

- **`EXTI0_1_IRQHandler`**: This is the interrupt service routine invoked for an external Pin 0 event. When it is invoked, the ISR **should disable the SysTick countdown timer (disable counter and the interrupt)**. It should then write the value EXTI_PR_PR0 to the EXTI_PR register to acknowledge the interrupt and clear its pending bit.
- **`init_rtsr`**: OR the value EXTI_RTSR_TR0 into the EXTI_RTSR register. This will set Pin #0 to generate an interrupt on the rising edge of a transition (button push).
- **`init_imr`**: OR the value EXTI_IMR_MR0 into the EXTI_IMR register. This will unmask the external interrupt for Pin #0.
- **`init_iser`**: Write the value (`1 << EXTI0_1_IRQn`) into the NVIC_ISER register. This will enable handling of interrupt 5, which is generated for a Pin 0 event.

Note that you will not be able use typical code like this to initialize a value for **`init_iser`**:

```
ldr r0, =NVIC
str r1, [r0, #ISER]
```

Since ISER is 0x100, it is too large for an immediate offset. Load the value into another register (e.g. r2), and use this instead: str r1, [r0, r2].

Four testing functions are provided to gradually test your assembly language subroutines. The first, **test_EXTI_1**, will check only that your ISR works correctly. You should see the green LED on the development board blink. The next three functions, **test_EXTI_2**, **test_EXTI_3**, and **test_EXTI_4** will test your RTSR, IMR, and ISER register initialization functions, respectively.

You do not need to demonstrate all of these functions. Only your code running with **test_EXTI_4**. The other functions are provided only for incremental debugging. If your **test_EXTI_4** worked correctly it will flash blue and the green LED out of sync.

Show your completed code to your TA.

6.5 Reaction timer

At the end of this section you will have created a simple reaction timer. The program will turn on the blue LED, and wait for the user to press the user push button. The two 7-segment displays show how many tenths of seconds passed between the LED turning on and user pressing the push button. It uses the push button as an EXTI interrupt to stop SysTick countdown timer when the user presses the push button.

For this section, you will need to complete **send_data(int data)**, which will "bit bang" the value of **data** bit by bit from LSB to MSB into PC2, while also generating a bit banged serial clock. Bit banging is *embedded speak* for toggling the output depending on the current bit. As an example consider 0b1001 0010, bit banging this value will set PC2 output to the following values logic 0, logic 1, logic 0, logic 0, logic 1, logic 0, logic 0, logic 1. This is bit banging from LSB to MSB.

`send_data()` is a simple bit banging subroutine. You will need to use "setpin and clrpin" from your previous lab. The two shift register put together can shift and latch 16 bits at most. So when we bit bang the lower 16 bits of "data".

The shift registers also need a clock (SRCLK) signal. This too will need to be set using bit banging. In the case of a clock, bit banging is a set of alternating 0's and 1's.

Here is pseudo code for **`send_data():`**

```
for_loop: 16 times
      if(data's lsb is 1)
            set_pin(GPIOC, PC2)
      else
            clr_pin(GPIOC, PC2)

      // Create bit-banged serial clock
      set_pin(GPIOC, PC0)
      nano_wait(200)
      clr_pin(GPIOC, PC0)
      nano_wait(200)
      right shift "data" by 1
      check loop exit
      branch to for_loop
```

To test and run the reaction timer, uncomment **reaction_timer()** function in main(). The send_data() subroutine will be used by reaction_timer() function to send the tick_count values into shift registers, so that it displays how many tenths of seconds have passed since the LED lighting up.

Show your work to your TA.

6.6 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.