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	10
6.2	Wiring	5
6.3	Implement a SysTick handler	10
6.4	Implement a GPIO interrupt handler	10
6.5	Reaction timer	5
6.6	Switch EXTI0 from PA0 to PB0	5
7.0	Post lab submission	*
Total		**65**

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

2.0 Objectives

1. Practice assembly language techniques
2. Learn about the SysTick and External GPIO interrupts of a microcontroller
3. 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

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 want to model a case where the CPU has something important to work on rather than just sit idly. This is modeled by a simple, slow division algorithm. The *slow division* algorithm is described in C on page 3, convert this to assembly. Type your code in lab4.s and test it by uncommenting the **test_slow_division()** call in main.c. If your program works correctly, it will flash the green LED. If you have made a mistake it will flash the blue LED.

Note: **test_slow_division()**, 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 *slow division* to work, you may take a zero score on this section, and replace the *slow division* call with micro_wait(2500000). You will need to add the micro_wait code into lab4.s.

Show this to your TA.

```
unsigned int slow_division(unsigned int numer, unsigned int denom)
{
    if (denom == 0)
        return 0;
    int quotient;
    for(quotient = 0; numer >= denom; quotient++)
        numer = numer - denom;
    return quotient;
}
```

Hint: Remember the difference between bls and ble? Which of them works with unsigned representation?

6.2 Wiring the micro to AD2

For the sections succeeding 6.2, it is key that you are able to visualize the outputs of the microcontroller. To visualize the outputs from the micro-controller we will use the AD2. The following connections are to be made. Connect PC0, PC1, PC2, PC3, PC4, PC5, PC6, PC7 and PC8 to DIO0, DIO1, DIO2, DIO3, DIO4, DIO5, DIO6, DIO7 and D15 of AD2, respectively. Configure them as inputs (LED) in AD2 (refer to lab 0 for a more detailed guide on configuring them as inputs). Connect the ground of the AD2 to the ground of the microcontroller.

Make sure the power to your microcontroller is disconnected 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). Connect your microcontroller to your development machine, and uncomment only the test_wiring() function in main(). This function is provided to you. You should see the "virtual" LEDs light up in sequence from D0 to D7.

Show your work to your TA.

6.3 Implement a SysTick handler

By the end of this section you will have a working binary counter that counts up every second, visualized using the AD2. To make that happen, you must complete the two subroutines **init_systick** and **SysTick_Handler** in lab4.s.

init_systick: This subroutine should initialize the SysTick down counter and reset value. It should set the counter to use the CPU clock. The repeat interval should be set to 31.25ms. Assume a 48MHz CPU clock.

SysTick_Handler: The C program on page 4 describes the functionality of SysTick_Handler. Convert this to assembly and type your program in lab4.s. The variables **seconds** and **tick_counter** are defined in main.c.

```
void SysTick_Handler()
{
    tick_counter = tick_counter + 1;
    if ((tick_counter & 0xf) == 0xf) {
            seconds++;
            display_led(seconds);
    }
}
```

To test your program, you should uncomment only the **test_SysTick()** call in **main().** It will call your initialization function and then continually run **forever_divison()** which will toggle D15 (and the blue LED on the dev board) after every calculation of slow_divison(40000000, 6). The interrupt will update LEDs D0-D7 counting up, while the forever_division will toggle D15.

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 or Pin 1 event. When invoked, the ISR should increment a **global variable "button_presses"**, then call display_led with button_presses as the argument. 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, if your program is correct. If your program is incorrect you should see the blue 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_5**. The other functions are provided only for incremental debugging. If your **test_EXTI_5** worked correctly, then upon pressing the blue button on the dev board, the LED count on the waveform window should go up by one.

Show your completed code to your TA.

6.5 Reaction timer

At the end of this section you would have created a simple reaction timer. The program will turn on the blue LED and wait for the user to push the button. The counter visualized using AD2 will keep counting up to let the user know how many milliseconds have elapsed. The counter stops as soon as the user pushes the push button. It uses the push button (the blue button on the development board) as an EXTI interrupt to stop SysTick countdown timer.

For this section, you will need to make the following changes:

- Change the SysTick timer to fire every 10ms.
- Comment the previous code of SysTick Handler and implement the code below in assembly

```
void SysTick_Handler()
{
    tick_counter = tick_counter + 1;
    display_led(tick_counter);
}
```

- Modify EXTI0_1_IRQHandler so that SysTick is stopped when the EXTI0_1_IRQHandler is invoked and comment the call to display_led.

To test and run the reaction timer, uncomment **reaction_timer()** function in main(). The LED visualization on the waveform tool will display how many hundredths of second (in binary) have elapsed since the LED lit up.

Show your work to your TA.

6.6 Switch EXTI0 from PA0 to PB0

It is easiest to use port A, pin 0 as the input for a external interrupt 0 because the port multiplexers are configured to use port A by default. If you want to use another port, there are a few non-obvious steps to take. The course staff want to make sure you know what they are (because it took us far too long to figure out how they work).

First, the port multiplexers for the EXTI inputs are controlled by the System Configuration Controller (SYSCFG) subsystem. You must enable the clock for the SYSCFG subsystem by updating the RCC's Advanced Peripheral Bus second enable register (APB2ENR). The bit you need to set is the same one that turns on the clock to SYSCFG as well as the comparator (see the Family Reference Manual, chapter 15). We have no idea why this single bit enables the clock for two two completely disparate subsystems, or why its use is so obscure. You should look up the documentation for the RCC's APB2ENR in the Family Reference Manual.

Once the SYSCFG clock is enabled, you must configure the four-bit field corresponding to the port multiplexer for pin 0 to change it from port A (the default, 0), to port B (1). To do so, you should first clear the four bits with BICS, then ORRS the value 1.

Finally, you must enable Port B by enabling its clock as you did last week. By default, its pins are inputs, so you need not do any more than turning on the clock.

Pseudocode for the operation is repeated in the comments for **enable_pb0()**. Complete **enable_pb0()**, uncomment the call to it in reaction_timer(), and rerun the reaction timer program to test it. This time, the user pushbutton does nothing. Now, you must provide an input on PB0. To do so, connect the AD2's DIO 8 wire to PB0. Configure DIO 8 in the StaticIO tool to be a button (Released: 0, Pressed: 1).

Show your work to your TA. He or she will verify that the user push button no longer does anything and that the StaticIO interface is the sole source of the input.

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.