Lab 3 Pre-lab

This lab will require you to program processor-specific items, so it is necessary to have the Arm-specific documentation present. We will also need the schematic and the reference manual, per usual.

Below are the key terms that should be useful for this lab and many others.

Key Terms

Polling - The idea of repeatedly checking a condition and then doing other stuff. Once this condition is met, you will do the appropriate things and return to doing other things.

Interrupt - This is different from polling. Interrupts are hardware-dependent and signal the processor when a certain condition(s) are met. Similar to polling, once those conditions are met, we do the appropriate things, BUT there are some limitations on what we can do that will be described later. Also, polling and interrupts are different but are not opposites of one another.

Compile Switches - The developer implements these to signify what code should be compiled and what shouldn't. This can be useful for many reasons. One reason is that it helps reduce memory consumption by not compiling functions that are not needed. It can also prevent functions from getting used when they shouldn't. In industry, you may have multiple products or variants of products residing on the same codebase. Therefore, you may want to compile switches so that Product A doesn't have the code for Product B and vice versa. Keywords for compile switches needed for this lab are #if, #elif, #else, and #endif. Consider the example below:

```
#include <stdio.h>

#define USE_SPECIAL_FUNCTION        0   // We will change this number accordingly

#if USE_SPECIAL_FUNCTION == 1 // Checks if the USE_SPECIAL_FUNCTION is equal to 1
int specialFunction(int i)
{
        return i + 2;
}

#else
int notASpecialFunction(int i) // If USE_SPECIAL_FUNCTION != 1
{
        return i;
}
#endif

int main()
{

        int numberForInput = 5;
        int output;
```

```
#if !USE_SPECIAL_FUNCTION  // This checks if it is set to 0.
       output = notASpecialFunction(numberForInput);
#elif USE_SPECIAL_FUNCTION == 1     // Similar to above, checks if USE_SPECIAL_FUNCTION is
equal to 1
       output = specialFunction(numberForInput);
#else
#error "USE_SPECIAL_FUNCTION must be 0 or 1"  // This verifies USE_SPECIAL_FUNCTION 1 or 0
#endif

       printf("Output from executed and compiled function: %d", output);
       return 0;
}
```

This would output:

```
Output from executed and compiled function: 5
```

If we changed USE_SPECIAL_FUNCTION to 1, we would get:

```
Output from executed and compiled function: 7
```

Lastly, we changed USE_SPECIAL_FUNCTION to a value, not 1 or 0, like 592, we would get a compilation error like the following:

```
Compilation failed due to following error(s).
main.c: In function 'main':
main.c:37:2: error: #error "USE_SPECIAL_FUNCTION must be 0 or 1."
   37 | #error "USE_SPECIAL_FUNCTION must be 0 or 1."  // This verifies USE_SPECIAL_FUNCTION 1 or 0
      |  ^~~~~
```

Function like Macros - These macros aren't "functions" but can take in an input "argument." Consider the example below where we have a macro that will give us shifted bitmask based on the input argument

```
#define SHIFT_BIT(x)    (1 << x)
```

Note - The following example will show an example interrupt configuration of an interrupt you will not use in the course. The purpose of this, similar to the last lab, is to give you an example of how things should be. We will use the external line 4 interrupt (EXTI4)

Nested Vectored Interrupt Controller (NVIC) - It controls interrupts and will give you the interrupt's name, position, and priority. You will need to see the table in the reference manual to know which position number is needed for the specific number.

5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable	DMA1 Stream0	DMA1 Stream0 global interrupt	0x0000 006C

Interrupt Numbers - These are the position of the interrupts in the NVIC and will need to be used for controlling the interrupts. EXTI4 is interrupt number 10 and has a default priority of 17.

Interrupt Flag Registers - The Cortex M4 Generic User Guide contains a section about the NVIC register summary. This section is important as you must know where these registers reside in memory and what each register does. Below will be a high-level explanation of some relevant registers we will need in this lab and some examples of interacting with them.

Interrupt Set-Enable Registers - This will enable a specific interrupt

```
NVIC_ISER0 |= (1 << 10) // Sets the interrupt in the 10th position
```

Interrupt Clear-Enable Registers - This will disable a specific interrupt

```
NVIC_ICER0 |= (1 << 10) // Clears the interrupt in the 10th position
```

Interrupt Clear-Pending Registers - This will clear the interrupt flag, which will take it out of the "pending" state

```
NVIC_ICPR0 |= (1 << 10) // Clears the pending flag of the interrupt in the 10th position
```

Interrupt Priority - The interrupt priority is very important when you have multiple interrupts enabled. The lower the number, the higher the priority is. Let's say we have two interrupts, interrupt A and interrupt B. Interrupt A has a higher priority. If these two interrupts happen at the exact same time, interrupt A will get serviced first. If interrupt B is being serviced and interrupt A happens, interrupt B will "pause," and Interrupt A will get immediately serviced, However, if interrupt A was getting serviced and then Interrupt B happens, Interrupt A would actually finish its routine and then interrupt B will execute after Interrupt A. Understand this concept, we will ask more about this in the midterm.

Interrupt Priority Registers - These will configure the priority for given interrupts. Note that an interrupt can have a non-configurable priority; this is usually true for very low system interrupts like bus errors and memory-related faults.

Interrupt Service Routine (ISR) - This code executes when the interrupt is triggered and serviced. Some things to note about these interrupt service routines, they don't return or take in values. They shouldn't be treated as normal functions. ISRs should also be kept as short and as simple as possible. It is bad practice to poll in an interrupt service routine or take an absurd amount of time to do things. It is not uncommon for other interrupts to be disabled or "masked" during execution of an ISR. This can be done at the developer's discretion. These are also referred to as IRQ Handlers.
For example, the ISR for the EXTI4 Interrupt would be

```
EXTI4_IRQHandler(void)
```

The process of configuring and using interrupts for GPIO (high level)

Recall - EXTI4 is the example interrupt

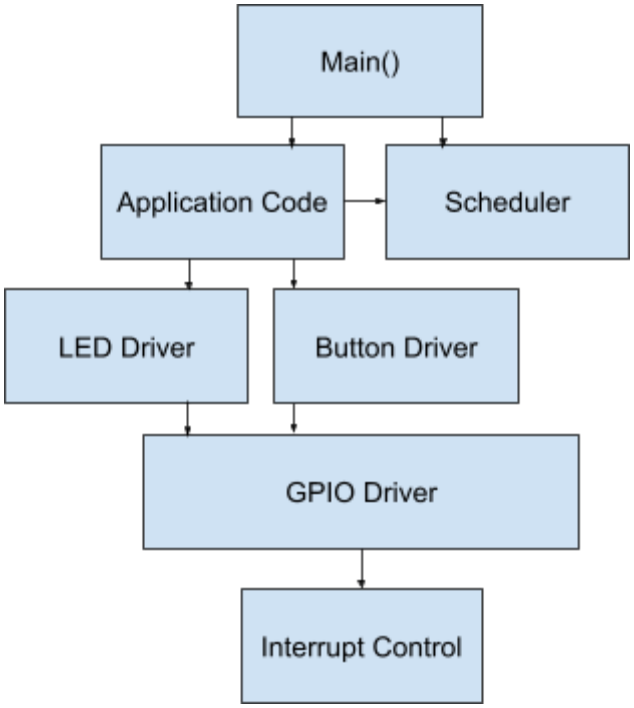
1. First, determine the interrupt number (interrupt position) of the interrupt you want from the reference manual
    a. EXTI4 is 10

2. *Given that this is EXTI4, this is for pin 4 on all GPIO ports. EXTI2 would be the for pin 2 on all GPIO ports. So that number signifies the pin. However, we need to specify which port should use this pin, as all ports cannot use this pin simultaneously.*
3. Configure the EXTI registers to match the interrupt functionality
    a. If you want a rising edge interrupt, you would access and enable the 4th bit (because we are using pin 4) in the RTSR register
    b. If you wanted the falling edge of the interrupt, you would access and enable the 4th bit in the FTSR register
    c. If you wanted both, you would enable both
4. Configure the SYSCFG to (The external interrupt configuration register specifically) to tell the system that you want a specific EXTI (External interrupt line) to be “configured” for a specific port
    a. You will write a 4-bit value to this register to signify 1 of the 16 ports
    b. Read the documentation and read the register map to get a better idea of what this entails. Don’t worry, the lab will go more in-depth.
5. Define and populate the EXTI4 ISR, and do the following in typically every ISR you create
    a. Disable the current interrupt you are in (so we would disable interrupts for EXTI4)
        i. You may want to disable additional interrupts here, but that is application specific. Next lab, we will get into multiple interrupts.
    b. Lower the pending flag of the specified interrupt
        i. You can do this by lowing the pending flag bit in the EXTI register OR the NVIC clear-pending register
    c. Do what you need to do in the ISR
    d. Re-enable all interrupts you disabled when you first entered the ISR
6. Determine which bit controls this interrupt within the NVIC register sets
7. Enable the appropriate NVIC interrupt
8. Set the appropriate bit in the EXTI register.

Setting up interrupts for different peripherals like timers or communication protocols are very similar but will use different registers. At the end of the day, this all embedded C, just us setting and clearing bits in registers and comparing those values and calculating other values.

Coding Hierarchy for Lab 3

Remember, files can only directly include files they “point” to in the diagram.


Main()
Application Code
Scheduler
LED Driver
Button Driver
GPIO Driver
Interrupt Control