Lab 3: Inputs and Interrupts
Due Date: October 4, 2024 @ 11:59 PM
Estimated Complexity: Medium
Estimated Time to Complete: 2 weeks

(Hypothetical) Engineering Request: An Aviation company wants you to help provide a proof of concept for their overspeed testing mechanism. During a flight precheck, they would like the pilot to be able to press and hold the button and the overspeed sensors are artificially fed speeds that will eventually exceed the “overspeed” threshold. The company wants you to provide firmware that supports polling and interrupting implementations. It will turn on an LED when the button is pushed. Compile switches should separate the implementations between polling and interrupts so the integration engineering team can easily switch between the two for risk assessment.

Prerequisites:

- Attend and understand the lab lecture
- Knowledge of compile switches and how to write them
- Knowledge of interrupts
- Knowledge of polling
- How to configure GPIO
- How to write an upper-level driver
- How to read the documentation

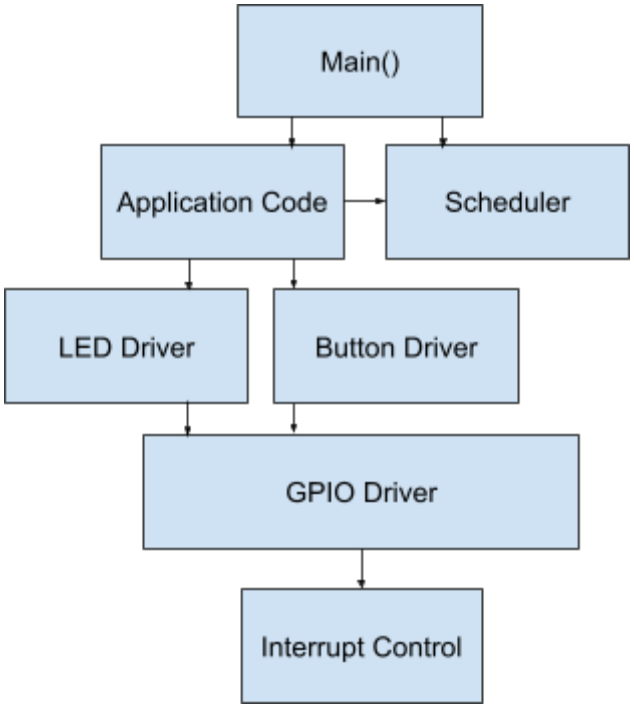
High-Level Overview:
This lab will be divided into two parts: Part A and Part B. Both parts will be functional only if the user button on the board is pressed. We will configure things so that when the button is pressed, the Green LED will turn on, and when it is released, it will turn off.

Part A will use polling implementation. In our super loop, we will call a series of functions to check the value of the port and pin responsible for the button. Given that information, we will either turn on the LED or turn it off. We will also examine the behavior of the delay function and how it relates to our system.

Part B will be using interrupt implementation. That is, we will NOT be checking the value for the port and pin responsible for the button. Rather, we will continue to conduct “normal business,” but once the button is pushed, we will handle the LEDs accordingly. We will also examine the behavior of the delay function and how it relates to our system.

Compile switches will control both parts and the delay function.

Coding Hierarchy:



Part A - Polling implementation

Lab Instruction:

1. Verify you have archived the previous lab (Lab 2) in a safe, valid location
2. Copy or Rename your project
    a. *Either should work, but if you rename your project, verify that you archived your previous lab in a safe location.*
3. Create the following fileset (header and source):
    a. Button_Driver.*
4. Create the following prototypes:
    a. In GPIO_Driver.h
        i. *Remember - All functions within GPIO Driver should have 'GPIO_' prefixed to the function; failure to abide by this will lead to a significant point deduction.*
        ii. One prototype to read from an input pin of a given port
            1. This will take in a pointer argument of type 'GPIO_RegDef_t' and a uint8_t corresponding to the pin number. This will return a uint8_t
        iii. One prototype to return the port number
            1. This will take in a pointer argument of type 'GPIO_RegDef_t' and return a uint16_t
            2. *Note, this function will not be immediately used in the polling implementation but can and can be added here now*
    b. In Button_Driver.h
        i. Find the port and pin number for the **User Button**. You can use the electrical schematic for this

ii. Create four (4) macros
    1. A macro for the Button's port value
    2. A macro for the Button's pin number
    3. A macro dictating if the button is pressed
        a. This can be 1 or 0. The choice is yours!
    4. A macro dictating if the button is not pressed or unpressed
        a. This can be 1 or 0 but needs to have the opposite value of the pressed macro
iii. Create the following prototypes, If a return type or an input argument is not stated, assume no return type or input argument
    1. A prototype to initialize the button
    2. A prototype to enable the clock
        a. You can make a prototype to disable the clock, but that function may never get used
    3. A prototype that will return a boolean that will return true/false if the button is pressed or not

c. In Application_Code.h
    i. Create the following prototypes
        1. An application-level button init function that will return nothing and will not have any input arguments
            a. This will be similar to our application LED init functions
        2. A function that will execute a routine if the button is pressed or not
            a. You can name it "executeButtonPollingRoutine" or something similar
            b. This will not take any input arguments and will not return anything

5. Add microcontroller specifics in STM32f429i.h:
    a. Create the macro for the port's base address responsible for the button
        i. You can place this directly above or below the port base address macro used for the LEDs
        ii. This will be one of the GPIO_ ports.. Where _ denotes a letter from A to K
    b. Create two new macros that act as functions to enable or disable a clock
        i. One will be for enabling clocks in the AHB1ENR register
            1. This will have an input "argument" corresponding to the correct bit of that register
        ii. One will be disabling clocks in the AHB1ENR register
            1. This will have an input "argument" corresponding to the correct bit of that register
    c. Create two macros that will be used for the macros above. They will correspond to the offset needed in the AHB1ENR register to enable or disable a specific clock
        i. One macro will be for GPIO port G and will get evaluated to the bit that corresponds with GPIOG. You can find it in the reference manual.

        ii. One macro will be for the GPIO port responsible for the button and will be evaluated to the bit that corresponds with that port. You can also find it in the reference manual.
    d. Delete the clock functions that were used to enable and disable the clocks for GPIO port G, these functions were created last lab
        i. We are removing these functions to make the code more modular and reduce having to write more macros. Instead, we will deal with the offsets to access and interact with certain bits in the register
6. Add appropriate code to the GPIO driver
    a. In the function that will be responsible for reading the value of the input pin
        i. Capture the value of the given pin number value from the input data register
            1. Instead of shifting left, shift the contents of the register to the right by the pin number, then do a bitwise AND with 1. It should look similar to the following:
            `Val = ((pGPIOx->REG >> p# ) & 0x01)`
            2. *Notice how before, in other functions, we would shift to the left. We shift to the right here for the sole reason of utilizing the AND operation. We do not care WHERE the bit is high; we just want to know IF it is high. Therefore, returning the bit location via a bitmask (for example, 0x01000) doesn't tell us much. If we did return that bitmask, we would still have to shift to the right to capture if the specified bit is high or low. If you have more questions about this, ask an instructional staff member.*
        ii. Return this value
    b. In the function responsible for clock control
        i. Replace the old clock enable/disable macros with the updated "function-like " macros we just created, and be sure to use the correct offset(s).
7. Create the Button driver code
    a. In the function responsible for initializing the Button
        i. Configure the button in a similar way you configured the LED, There are a couple of differences to keep in mind
            1. Since we are creating a local struct (rather than a static one) for the button configuration, it would be smart to initialize it to zero when declaring it.
                a. *This wasn't needed for the LED because we declared them as static, and one of the C properties of a static variable is that it gets zero-initialized automatically (we will ask you about this concept on a midterm or quiz) ;)*
            2. Use the macros created in the header file for the correct port and pin number of the button

            a. *Given this is the only button, this is why this function does not have to be as modular as it would be if we had multiple buttons*
        3. Since this is a button, we want to ensure this is configured in input mode (as opposed to output mode like the LED was)
        4. Enable the clock for the button by calling the appropriate function in the button driver
            a. *Technically we haven't populated that function yet, but you can still call it.*
        5. Call the GPIO init function and pass the local struct you made by reference
    b. In the function responsible for enabling the clock
        i. Call the function from the GPIO driver responsible for enabling the clock with the proper arguments
            1. Use the macro for the button port you should've created in the header file
    c. In the function responsible for determining if the button is pressed or not
        i. Using the GPIO driver, read the input of the button's pin and compare the return to the Button Pressed macro you created earlier
            1. *It may be good to verify that the Macro you created for if the button is pressed, makes sense. You can change it if needed.*
        ii. Return true if the button was pressed, false otherwise
8. Add appropriate code to the LED driver code
    a. Configure and initialize the Green LED in the function responsible for initializing the LED. The configuration should look very similar to the other LED, besides it will just have a different pin number
9. Add the appropriate code to the scheduler code
    a. In the header file
        i. Add a new event that will poll the button
            1. Remember to shift it one more bit to the left than the previous event
10. Add appropriate code to the application code
    a. In the application init function
        i. Call the LED init function of your newly configured Green LED
        ii. Deactivate the LED (We want the pin (the LED) to be in a known initial state)
        iii. Add the button polling even to the scheduler
        iv. Verify the delay event is added to the scheduler
            1. This code was created in a prior lab
    b. In the Button initialization function
        i. Call the button initialization function created in the button driver
            1. *Recall, there should NOT be an input for this function*
    c. In the function responsible for executing a routine when the button is pushed
        i. Check if the button is pushed down or not,

1. If the button is pushed down, turn on the Green LED
    a. You can use a switch or an if statement for this logic. The choice is yours!
2. If the button is not pressed, turn off the Green LED
    a. Again, you can use a switch or an if statement for this logic. The choice is still yours

11. Integrate code to main()
    a. Similar to the delay function, add a check to verify if the events to run are equal to the button polling event
        i. If it is, call the execute button polling routine you created in the application code
12. Build and download the code to the board and debug
    a. There should be no warnings or errors
13. Give it several tries and try to push the button. If needed, hold it down to verify that the light does come on eventually.
    a. *If your button seems to "sometimes" work and toggle the LED, that is a good thing in this case… our implementation of what we have is not the best…in Part B, we will improve things drastically*
14. Consider the following questions; they may show up on a quiz.
    a. What do you notice about the "response" of the button?
    b. If we remove the delay function (or comment out the scheduler call that adds the delay event), does the button's response change?
        i. You must rebuild and redownload the code for this

Part B - Interrupt implementation

Lab Instruction:

1. Create the following fileset
    a. InterruptControl.*
2. Add processor interrupt details
    a. In STM32f429i.h
        i. Add macros for the NVIC registers
            1. First, locate the addresses of the NVIC registers from the Cortex-M4 manual
                a. You only need to worry about the first register so NVIC_ISER0, NVIC_ICER0, etc...
            2. Then for the set-enable, clear-enable, set-pending, and clear-pending, create macros for the base register (for example, for set-enable, it would be NVIC_ISER0)
            3. Cast the address of a pointer type of uint32_t, make sure this is volatile
                a. For example, for Set-Enable:

```
#define NVIC_ISER0   ((volatile uint32_t*) 0xE000E100)
```

        b. *This is no different from what we do with things like the GPIO_RegDef_t. We just don't have a register definition here so we use unit32_t.*
    4. *We are not making a register map struct like we did for GPIO_RegDef_t because there are a significant number of NVIC registers that are not available on the microcontroller.*
- ii. Locate the base address of the SYSCFG, the EXTI peripheral, and the bus that both peripherals reside on
- iii. Make three macros, one for each of the base addresses
    1. The SYSCFG and EXTI base addresses should reference the APB2 base address macro. This is similar to what we did with the AHB1 base address and the GPIO peripherals ie:

```
#define GPIOG_BASE_ADDR     (AHB1_BASE_ADDR + 0x1800)
```

- iv. Create a register map struct for the SYSCFG register set
    1. Make the Control registers for the EXTI lines an array within the struct rather than four different members
    2. *Remember, a struct can also have an array as a member element of the struct.*
- v. Create a register map struct for the EXTI register set
- vi. Create the macro needed to interface with the SYSCFG register, similar to what we did with GPIOA, GPIOG, and RCC
- vii. Create the macro needed to interface with the EXTI register, similar to what we did for SYSCFG
- viii. Create a clock-enable macro to enable the SYSCFG peripheral clock
    1. You can find out which offset and the RCC register you need from the reference manual
- ix. Create a clock disable macro to disable the SYSCFG peripheral clock
- *x. Note we do NOT need to enable the clock for the EXTI due to its functionality. It doesn't have a "clock" that we can just enable and disable*

3. Add general interrupt control prototypes, macros, and functions
    a. In InterruptControl.h
        - i. Determine which IRQ number would be responsible for the Button and create a macro to reflect that value
            1. You can name it something like EXTI*_IRQ_NUMBER where * denotes the proper interrupt name
        - ii. Create the following prototypes; these will all take in a uint8_t argument that will be the IRQ number and return nothing. The naming of these prototypes should contain 'IRQ' somewhere in the name.
            1. A prototype to enable an interrupt
            2. A prototype to disable an interrupt
            3. A prototype to clear a pending interrupt
            4. A prototype to set a pending interrupt
        - iii. Create a prototype to clear the interrupt pending bit in the EXTI interrupt register

1. This will return nothing and take in an argument of uint8_t representing the pin number

b. In InterruptControl.c
    i. In the 4 functions are responsible for enabling interrupts, disabling interrupts, setting pending interrupts, and clearing pending interrupts. Do the following:
        1. If the IRQ number is less than 32, set the proper bit in the proper NVIC register by shifting a bitmask by the IRQ number
            a. There is some pointer logic you will need to be mindful of here
                i. Hint: how do we put a certain value at a certain memory address?
            b. Revisit the lab lecture if you need to be refreshed on why we care about the IRQ number here
        2. If the Irq number is greater than or equal to 32, we should set the appropriate bit in the proper NVIC register by shifting the bitmask by the module of the IRQ number and 32
            a. Recall the purpose of Modulo
    ii. In the function responsible for clearing the pending EXTI interrupt
        1. Set the proper bit in the pending register of the EXTI peripheral by creating a bitmask using the pin number as the shift value
            a. You may be thinking, why are we SETTING a bit in the pending register to CLEAR the bit? Read the documentation

4. Add interrupt support to the GPIO driver
    a. In the header file
        i. Add a member to the PinConfig struct, this will be for the pin interrupt mode
        ii. Add four (4) macros to specify the following interrupt modes:
            1. No interrupt selected
            2. Falling edge interrupt
            3. Rising edge interrupt
            4. Falling and Rising Edge interrupt
        iii. Create macros to specify the port “number” for the button port
            1. For now, just know GPIOA would be 0, GPIOB would be 1, and GPIOC would be 2, etc... So make sure this macro evaluates to the proper number
                a. This will get used as an offset later on
        iv. Create a prototype that will return the port number *(if not already)*
            1. This will take in a pointer of type GPIO_RegDef_t and return a uint16_t
        v. Create a prototype that will enable or disable an NVIC interrupt for the GPIO peripheral

1. This will take in two arguments, one of uint8_t that will be for the irq number and one to determine if we will enable or disable an interrupt
2. This will return nothing

b. In the source file
    i. In the function responsible for initializing the GPIO
        1. Immediately after configuring the mode, we need to see if we have any other interrupt mode than “no interrupt mode” by using the new struct member added.
        2. Depending on the interrupt mode selected, configure the proper registers
            a. If the mode is configured to be FT (Falling Edge/Trigger), we need to set the proper bit in the FTSR register of the EXTI Controller
                i. Use the pin number to construct the bitmask
            b. We also need to clear the same bit in the RTSR register
                i. This may seem like an “extra” step. But this ensures that this value is what we want and expect it to be. In bigger systems, multiple clients can access the same register set as you, thus leaving them in a different state than you intended
            c. If the mode is configured to be RT (Rising Edge/Trigger), we need to set the proper bit in the RTSR register and clear the proper bit in the FTSR register
                i. Use the pin number to construct the bitmask
            d. If the mode is configured to be both the Rising Edge/Trigger AND the Falling Edge/Trigger, set the appropriate bits in the appropriate registers (RTSR and FTSR)
        3. We need to enable the appropriate bits in the system configuration controller peripheral
            a. Understand the relationship between the SYSCFG, EXTI, and IMR registers.
            b. Create a local variable of type uint8_t and set it equal to the pin number divided by 4
                i. This will be used to select the appropriate register
            c. Create another local variable of type unint8_t and set it equal to the pin number modulo 4
                i. This will be used to select the appropriate bit field
            d. Create another local variable of type uint16_t
            e. Call the getPortCode function with the appropriate port
            f. Set that return value to the uint16_t you just created.
            g. Enable the clock for SYSCFG

            h. Using the three variables you created, access the appropriate bitfields in the appropriate external interrupt configuration register
                i. Refer to the process we did in the last lab to configure the alternate function register
                ii. We will also need to multiply one of these variables by 4
        4. Then enable the interrupt delivery by enabling the appropriate bit in the Interrupt Mask Register in the EXTI controller
    ii. In the function responsible for returning the port number
        1. Based on the input argument, return the correct port number.
            a. If you'd like, you can also make this function return the port number for GPIOA
            b. This function may seem annoying or "useless," but this goes back to our coding hierarchy and not using magic numbers. Since the port number *is done at compile time, we can make it a macro, thus removing the chance of it being a magic number*
    iii. In the function responsible for configuring an NVIC interrupt for the GPIO driver
        1. Using one of the input arguments, determine if we are enabling or disabling interrupts, then call the appropriate function from the InterruptControl fileset and pass the other input argument into it
5. Add interrupt support to the Button driver
    a. In the header file
        i. Add a prototype that will initialize the button for "interrupt mode."
    b. In the source file
        i. In the function responsible for initializing the button in interrupt mode
            1. Create and populate the members of a local struct similarly to what we did for the original button initialization function. The only difference is we will configure the new struct member added responsible for the interrupt trigger mode. This will be set to use the falling and rising mode
                a. We want the LED to turn on when the button is pushed down and turn off when the button is released
            2. Enable the button clock
            3. Call the GPIO_Init function and pass the local struct by reference
            4. Call the GPIO interrupt configuration function and enable the proper interrupt
6. Add interrupt support to the Application code
    a. In the header file
        i. Add compile switch definition
            1. Make a macro named something along the lines of "USE_INTERRUPT_FOR_BUTTON" and set it to 1

ii. Create a prototype that is responsible for initializing the button for interrupt mode.

b. In the source file
   i. In the application initialization function
      1. Add compile switches
         a. The button initialization (interrupt support) should be called if the “use button interrupt” macro is not zero
            i. I use “ “ (quotes) because the name of this macro dictating this could vary from developer to developer
         b. The button initialization (non-interrupt support) and adding the button polling event to the scheduler should only be called if the “use button interrupt” macro is zero
      2. Verify the delay event is added to the scheduler
   ii. Add compile switches so that:
      1. The button initialization (non-interrupt support) and the execute button polling routine function definitions are only compiled and called when the “use button interrupt” macro is set to 0
      2. The button initialization (interrupt support) function is compiled and called if the “use button interrupt” macro is set to 1
   iii. At the bottom of the file, create a function definition that will be used as an interrupt handler
      1. This function **must** be named something very specific, refer to startup/startup_stm32f429~.s to find the appropriate interrupt’s name
      2. This must be named this, as it is the name of the interrupt in the NVIC, and the processor knows exactly where this interrupt handler is based on the NVIC
      3. We do not need to create a prototype for this function as it has already been “created” elsewhere. We are just “overriding” their definition
   iv. Within the IRQ handler
      1. Disable the interrupt using the disable interrupt function from the InterruptControl fileset and the Macro created that has this interrupt IRQ number
      2. **Toggle** the Green LED
         a. It may seem odd that we are toggling it rather than just turning it on or off, but think about it: The interrupt is configured to detect the rising edge AND the falling edge. This means that pushing and releasing the button will trigger the interrupt each time (so twice).
      3. Clear the appropriate pending bit in the EXTI register, and use the function you created to do this

        a. *This is very important and can lead to some "bugs" if you don't do this properly*
    4. Re-enable the interrupt using the enable interrupt function
7. Adjust code in main()
    a. Add compile switches
        i. You should not check for the button polling event nor run the polling function if we use the button interrupt
8. Build and download the code, debug if necessary
9. Consider the following questions; they may show up on the quiz
    a. What do you notice about the button response now?
    b. Does the delay function appear to do anything?

Acceptance Criteria:

- When using the polling implementation, there is a clear delay between the button and the GREEN LED's behavior
    - When the button is pushed down and held down, the GREEN LED eventually turns on.
    - When the button is released after holding it down long enough for the GREEN LED to turn on, the LED eventually turns off
- When using interrupts, there is no delay between the button and the GREEN LED
    - When the button is pushed down, the LED immediately turns on.
    - When the button is released, the LED immediately turns off.
- A compile switch adequately separates both implementations

Additional Notes:

- Using the generated HAL code is prohibited, if you use the HAL, you will forfeit ALL points for this assignment.

Grading Rubric:

This lab will be worth 150 points. The breakdown of grading will be given below.

1. Code Compilation (20 points)
    a. Full credit - Code Compiles with 0 errors and 0 code warnings
    b. Partial Credit - Code contains warnings (-5 points for each warning)
    c. No credit - Code does not compile (Student has at least one error)
2. Code Standards and Hierarchy (20 points)
    a. Proper naming of functions/files
    b. Proper layering of files
    c. For each violation, 5 points will get subtracted from the 20 points possible
3. Code functionality (45 points)
    a. Each Acceptance criteria point will be 15 points each
4. Project Exporting (5 points)
    a. Was the project exported right with the appropriate naming?
5. Lab Attendance (10 points)
    a. Did the student attend lab sessions appropriately and consistently?

6. Interview Grading (50 points)
    a. Does the student understand a basic concept utilized in their code? (15 Points)
    b. Does the student understand a semi-complex concept utilized in their code? (25 points)
    c. Does the student understand or partially understand a semi-advanced concept utilized in their code? (10 points)