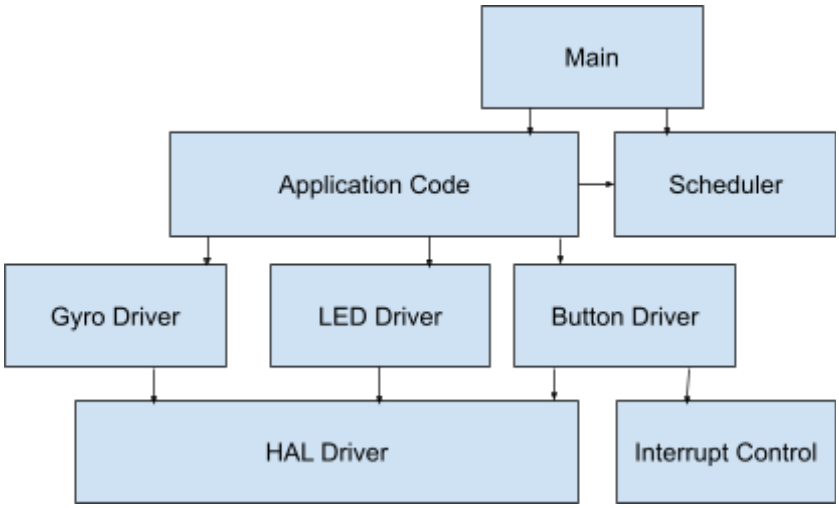
Lab 5: Introduction to HAL and using SPI
Due Date: 11/08/2024 @ 11:59 PM
Estimated Complexity: Medium
Estimated Time to Complete: 2 weeks

(Hypothetical) Engineering Request: You are tasked with helping a startup specializing in oceanic sensor development. They want you to provide code that can act as proof of concept for achieving sensor readings from their latest device. You are tasked with using the SPI communication protocol and getting two pieces of information from the device: the device ID and the temperature. You will get this information each time a button is pressed. You will use the STM32F429i microcontroller to write this code and communicate with the gyro/temp sensor on that board. You will be using the HAL (Hardware Abstraction Layer) to assist with your code and make it modular for the processor they will be using.

Coding Hierarchy:



Prerequisites:

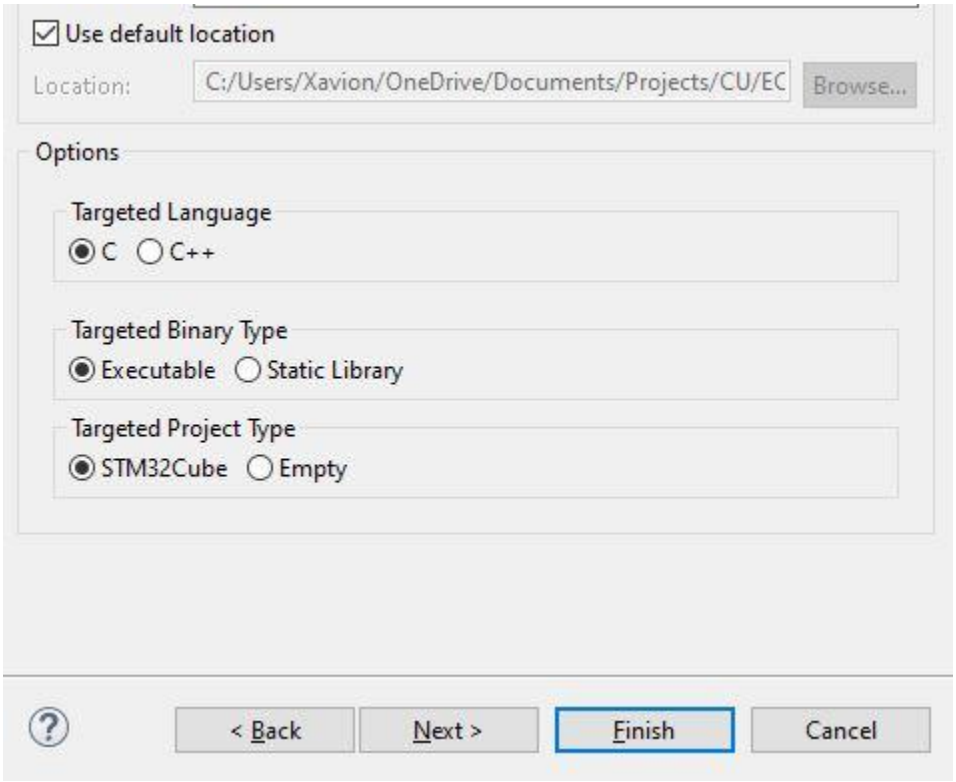
- Know how to read and understand documentation
- Solid understanding of the SPI protocol
- Somewhat comfortable with the idea of integrating code that you did not write with code that you did write
- Strong ability in reading code you did not write
- The ability to “trust” that code that you are supposed to leverage is doing what it is saying it does
- The ability to modify existing code to adapt to new architecture

Background information:
Congratulations! We will no longer have entire labs dedicated to you writing drivers from the ground up. Now, the next critical skill for being a programmer is integrating and leveraging code you did not write and had no say in how it was written. In the industry, code integration is an essential skill and ability that you need in your toolbox. This lab will use the HAL for the STM for

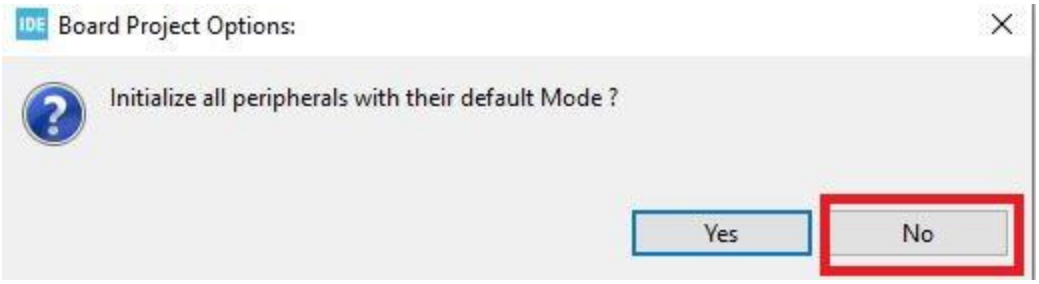
some drivers. The generated HAL code looks terrifying initially, but if you take it line by line, you will notice you've written similar code yourself! **Build your code periodically to resolve build errors as you go.**

Lab Instruction:

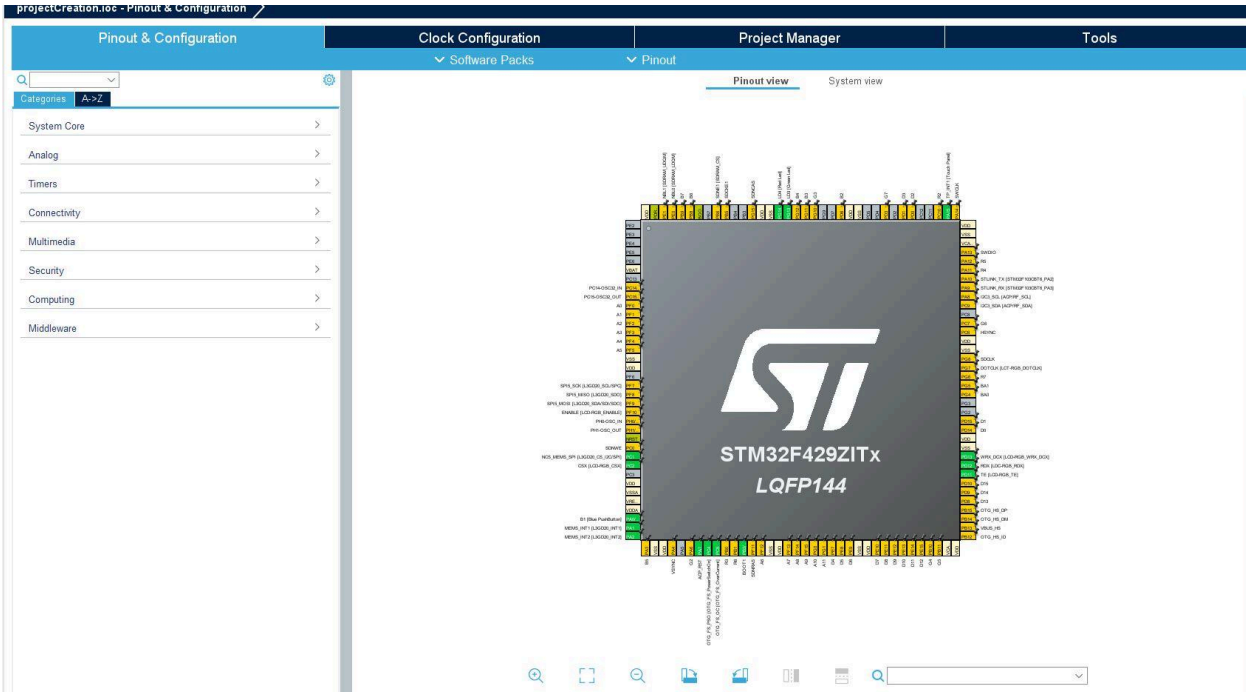
1. Verify you understand the lab lecture and can meet the prerequisites
2. For this lab, you should use some functionality from lab 3 to build off of, as we will not be utilizing the timer in this lab
3. Create a new project to bring in the HAL Driver for SPI and GPIO
    a. From the IDE,
    b. Go to "Board Selector," search and select our microcontroller, and click next
    c. In the 'Setup new project", name the project appropriately and select "STMCube32" for



    d. Click Finish
    e. When the popup asks you if you want to "Initialize all peripherals to their default Mode", **Select NO**
        i. Failure to abide by this will result in a massive amount of code being generated and all peripherals being enabled.



f. This screen should show up:



g. Select which peripherals should be enabled
   i. In the System Core section, ensure the NVIC and GPIO are selected
      1. This is the default, and they should already be selected “Green”
   ii. In the Connectivity, make sure you have the correct SPI selected
      1. *Refer to the electrical schematic and the document that contains the pinouts to determine which SPI is needed to interface with the GYRO*
      2. You must click the appropriate SPI and change the mode to ‘ Full-Duplex Master’
      3. We will trigger the “Chip Select” manually, so select ‘Disable” for Hardware NSS Signal

h. Configure the clocks
   i. Using the reference sheet for the Gyro and the properties of SPI, make sure the bus’s clock that is connected to the SPI we have is not going absurdly too fast or too slow
      1. You can technically adjust this value later, but you will need to do some digging to find the code that is responsible for that

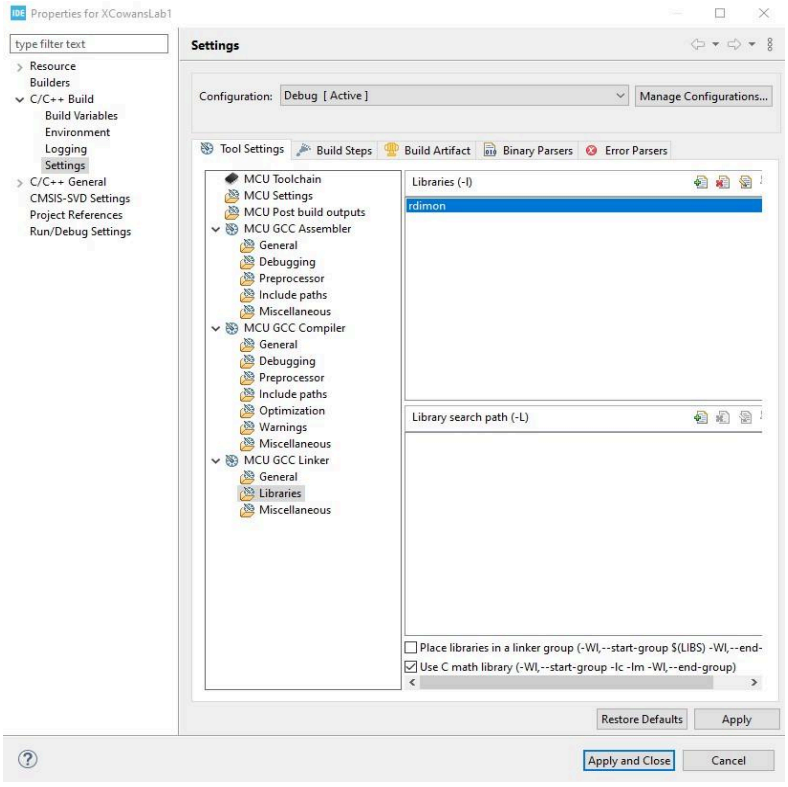
ii. You can use change the System Clock Mux to whatever you feel appropriate
    1. *HSI is the “default” clock in our microcontroller*

i. Go to Project > Generate Code
j. They will ask if you want to go to a C/C++ perspective; select yes.
k. Congratulations! You have generated your project using the HAL
l. Notice how the src and inc directories are in the Core directory

4. Configure printf functionality
    a. Delete Core/Src/syscalls.c (Right-click the file and go to delete)
    b. Add the following to main.c:

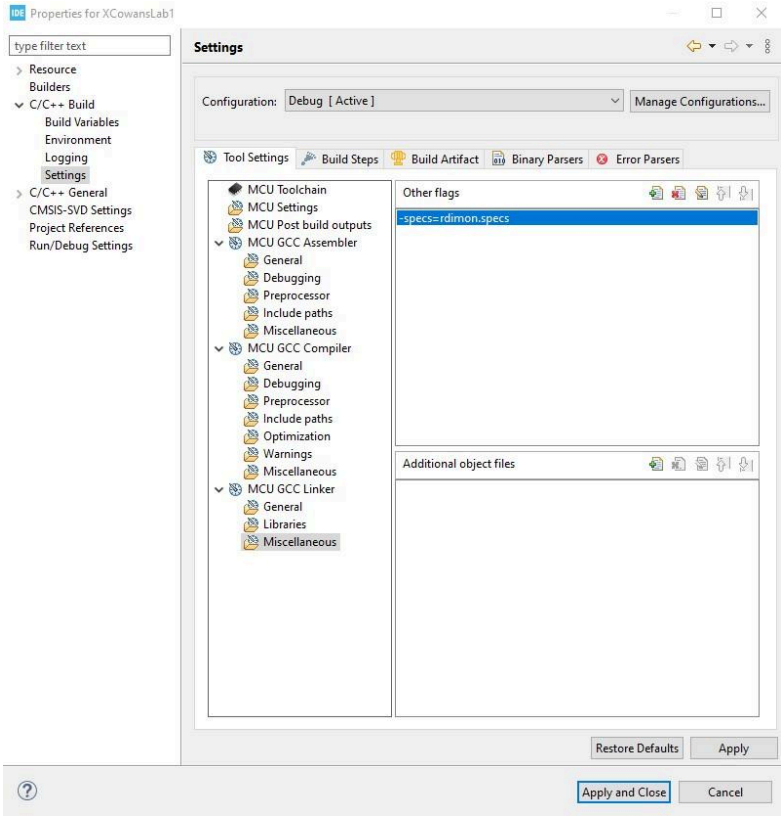
```
#include <stdio.h>
extern void initialise_monitor_handles(void);
```

    c. Add a function call to initialise_monitor_handles() as the first thing to get executed within main
    d. Update Linker configuration
        i. In Project > Properties > C/C++ Build > Settings > MCU GCC Linker > Libraries, add “rdimon”
            1. *This verifies the proper libraries and functions get linked when the code is compiled. Refresh on the C compilation process if needed*



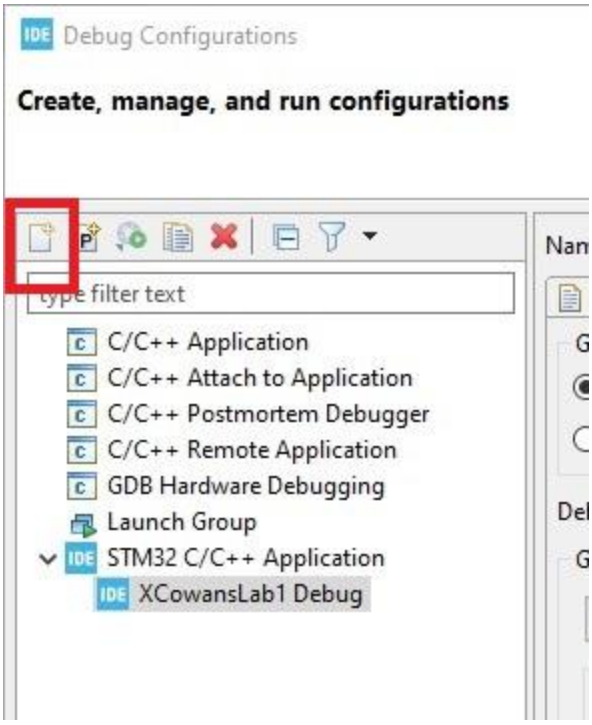
        ii. In the Miscellaneous section, add the flag “-specs=rdimon.specs” in the other flags section

1. *Compiler flags are used to tell the compiler to do or consider certain things. You will learn more about this in later courses*



iii. Click Apply and Close

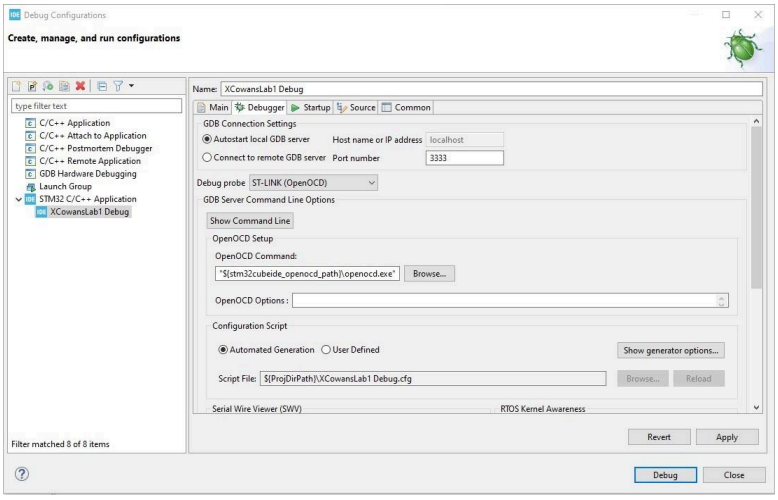
e. Build the project
   i. Just click the hammer icon in the toolbar
      1. Alternatively, you can go to project > build all
      2. Alternatively, CTRL + B
   ii. The project should build with 0 errors and 0 warnings
      1. *This means that the code was able to be successfully compiled, and machine code was able to be generated. This is good because then this means this can be flashed to the board!*
f. Create/Update OpenOCD debug configuration
   i. Select Run > Debug Configurations
   ii. Select STM32 C/.. and click the “New launch configurations”



1. Alternatively, you can also right-click STM32 C/C++ Application and select “New Configuration”

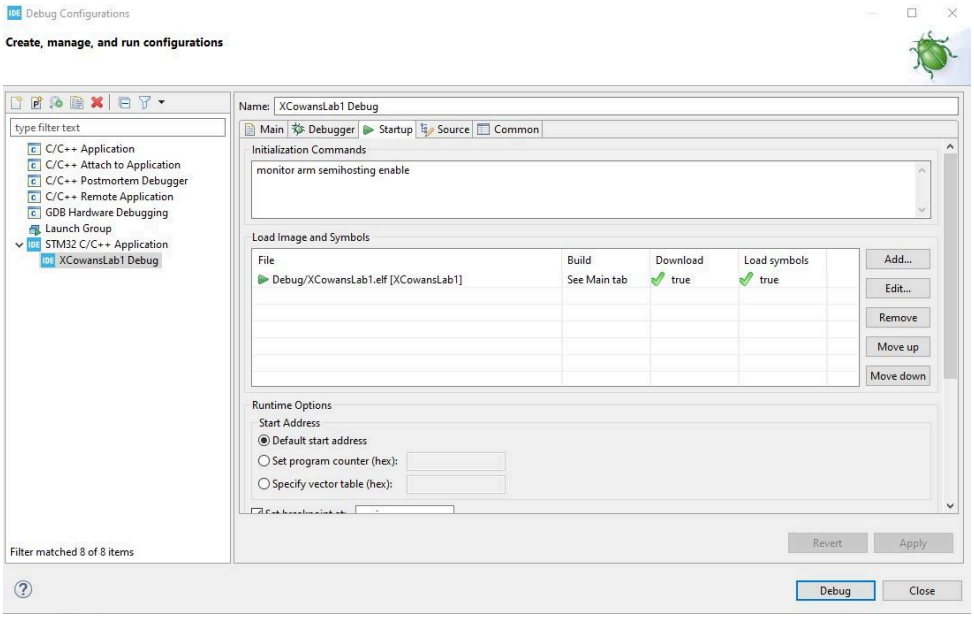
iii. In the Debugger tab, change “Debug Probe” to ST-Link (OpenOCD)

1. *This communicates with the board on which debug module we want to use. Many boards, like this one, only have one debug probe available*



iv. In the “Startup” tab, in the Initialization Commands, add “monitor arm semihosting enable”

1. *This tells the debug session (GDB) to use semihosting*



v. Click Apply
vi. Click Close

5. Integrate your source and header files (from lab 3) into the project
    a. You will include all filesets but the GPIO_Driver fileset
    b. DO NOT include your stm32429i.h file, the HAL uses its own version of that file, and it can be found in Drivers\CMSIS\Device\ST\STM32F4xx\Include\stm32f429xx.h
6. Change appropriate function calls
    a. We need to call the correct functions since we are using the HAL GPIO driver.
        i. This may also require you to change certain data types for them to be compliant with the HAL GPIO functions
    b. This includes, but is not limited to, Init functions and clock enable/disable functions
    c. Be mindful of macro uses, our macros were created for our drivers, they may not map 1 to 1 when using the HAL Driver
        i. For example, notice what we used for GPIO_PIN_0 in prior labs and what STM uses for GPIO_PIN_0 in their labs
7. Update file includes as needed
    a. If you don't want 200+ errors due to "redefinitions" and such, use the general header file
        i. "stm32f4xx_hal.h "
        ii. Examine that file and see why this file suffices as the only file to include to interact with the HAL.
8. Consider building your code and resolving the build errors if you can
9. Create and populate the ErrorHandling fileset
    a. The header should contain the function named APPLICATION_ASSERT, this should not return any but takes in a boolean argument

b. The source should contain the APPLICATION_ASSERT function, which goes into an infinite loop if the input argument is false
    i. It will be very similar to the ‘Error Handler’ function in the main.c generated by the HAL

10. Create and populate the Gyro fileset
    a. The header should contain the following:
        i. Macros for the Gyro
            1. These should be the addresses of needed registers on the Gyro
        ii. Macros for the Pin and Port information for the Gyro
        iii. The following prototypes:
            1. NOTE- All Gyro prototypes in the Gyro fileset should have “Gyro_” prepended to the function name.
            2. A prototype to Initialize the Gyro
            3. A prototype to get the device ID and print it
            4. A prototype to power on the Gyro
            5. A prototype to get the temperature and print it
            6. A prototype to configure the registers on the Gyro
            7. A prototype to read the registers of the Gyro
            8. A prototype to verify the HAL status for the SPI is okay
            9. A prototype to manually enable slave communication
            10. A prototype to manually disable slave communication
            11. Once you start writing the source code, there may be repeated code that will make you want to create prototypes.
        iv. You may need some macros later on. It will be your responsibility to add them later on
            1. Remember, we should not have magic numbers in the source code. Create Macros as appropriate.
    b. The source files should contain definitions for the prototypes mentioned above
        i. Read the reference manual and other relevant documentation to determine which SPI configurations are needed
        ii. When configuring the registers for the Gyro
            1. For the Control Register 1
                a. We want to have the power down mode to be in normal mode
                b. The bandwidth bit field *should* be okay to be left at 00
                    i. It may help to have it at 01
            2. For the Control Register 5
                a. We want the Reboot memory content to be enabled
        iii. Create a static variable to keep track of the HAL status
            1. That said, any HAL function that returns the status should be assigned to this variable
            2. The function responsible for verifying the HAL status is OK should be called regularly (when it is possible for the HAL status to

change) and should fall into the application assert infinite loop if the HAL is not okay

iv. You may want to create static variables for other things, and that is okay as long as coding hierarchies are not being violated.

11. Update the scheduler code
    a. You should add the following event flags if it is not already there
        i. APP_DELAY_FLAG_EVENT
        ii. DEVICE_ID_AND_TEMP_EVENT
    b. You do not need any other events besides the one above
12. Populate the Application Code
    a. Create the prototypes and functions needed to prevent main.c from calling into interfacing with the Gyro_Driver code directly
    b. When adjusting your button interrupt, remember, the interrupt should add the event responsible for printing the device ID and temperature to the scheduled events
    c. Add back/Create an application delay, you can do this however method you want
13. Clean up and populate main.c
    a. You can keep main.h, and it's include if you want
    b. You may leave the SystemClock_Config and its call in main.c
    c. You may also leave the HAL_Init call in main.c
    d. You may want to use HAL_Delay() right after your call to power on the Gyro to give the Gyro time to power on before trying to interact with it
        i. 100 ms should be plenty of time
    e. Delete the calls to MX_GPIO_Init() and MX_SPI_Init()
    f. Verify you added the necessary functions/libraries to be able to use printf
    g. Powering on the Gyro and configuring the registers for the Gyro should all happen before the super-loop
    h. Integrate the scheduler and call appropriate functions when the appropriate event flags are set
14. Build and debug
15. Verify that all acceptance criteria are met
16. Export Your project and turn it into Canvas

Acceptance Criteria:

1. You receive and print the **correct** Device ID (**IN HEX)** from the Gyro, shortly after button is pushed
2. You are receiving and printing consistent and *reasonable* temperatures (**IN DECIMAL)** from the Gyro shortly after button is pushed

Failure to print the data in the specific format will result in you receiving no credit for that Acceptrance Criteria. For example, if you put the device ID in decimal, you will forfeit the 25 points.

The term *reasonable* is put like that as the temperature module within the Gyro seems to be cost-efficient so the performance is not super ideal.

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 (-10 points for each warning)
    c. No credit - Code does not compile (Student has at least one error)
2. Code Standards and Hierarchy (10 points)
    a. Proper naming of functions/files
    b. Proper layering of files
    c. Any violation will result in all 10 points being lost
3. Code functionality (50 points)
    a. Each acceptance criteria is worth 25 points
4. Project Exporting (10 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)