MTE 325 - TWO AXIS MACHINE PROJECT

ALLYSON GIANNIKOURIS*, P.ENG.

Revised Winter 2025

Version 1.3

CONTENTS

ACKNOWLEDGMENTS

This project would not have been possible without both the ideas and financial support of Prof. Sanjeev Bedi and the IDEAs clinic. Chris Rennick is owed thanks for taking care of all the ordering. The IDEAs clinic co-op students from W18 and S18 assisted with the initial assembly of the machines, putting together the kits and taking care of the seemingly endless amount of wire crimping that was required to make it all work. Thanks are also owed to Chris Rennick and Andrew Milne for helping with evaluation of the pilot and being sounding boards for changes. They were especially helpful in revising the project structure and determining a suitable grading scheme. Thanks are also due to the MME lab staff, in particular Paul Groh and Richard Yan, for the maintenance and upkeep of the lab room and equipment.

* *Dept. of Mechanical and Mechatronics Eng., University of Waterloo*

1 INTRODUCTION

This project will provide an opportunity to apply course concepts to a real-world system. In particular we will focus on the aspects of embedded systems that are most relevant to this course: interfacing with and using peripherals. The project has been divided into several deliverables, which will allow application of course concepts as they are learned.

Students will work in groups of 3 to 4. Groups will be chosen by the students, with the restriction that all must be assigned to the same lab section. There are three machine types available this term. The groups on learn have been setup to reflect machine availability. Each group will have a lab kit containing a micro-controller and motor shield, bread board, potentiometer and various other discrete components. Groups will be able to sign out a kit for the term by filling out the sign-out sheet posted on Learn and bringing it to their first lab session.

1.1 Learning Objectives

After completing all deliverables in this project, you should be able to:

- Implement and compare interrupts and polling
- Use industry grade tools and equipment to configure and debug an embedded system
- Configure and use parallel ports in an embedded system
- Use serial interfaces in an embedded system
- Configure and characterize the behaviour of an analog to digital converter (ADC)
- Characterize the behaviour of two-axis machine driven by stepper motors in an embedded system
- Use common lab equipment such as an oscilloscope and multimeter to debug embedded systems
- Write C code that follows generally accepted industry standards for structure and formatting

1.2 Lab Sessions

Session 1: This session is dedicated to ensuring all groups have a working set of tools and hardware. You should review Section 2 before your lab session and install the necessary tools. Section 4 contains the information and exercises for this session.

Session 2: This session will be used to complete an exploration of synchronization techniques. Section 5 contains the information and exercises for this session.

Session 3-5: These sessions should be used to work on and demonstrate the 2-axis machine objectives. Section 6 contains the information and exercises for these sessions.

Session 6: This session should be used for any remaining demonstrations of 2-axis machine objectives and for testing and demonstration of the functional objective.

1.3 Tasks

This project is divided into multiple deliverables that are spread throughout the term. In total the project is worth 30% of your final grade. Submission-based tasks have rubrics associated with the dropboxes in Learn and the demonstration rubrics are posted in the Project module on Learn.

Task	Points	Bonus Points	Approximate Date
SOP Quiz	1	0	Before session 1
Tool Test	0	0	Session 1
Polling and Interrupts	6	0	Session 2
Limit Switches	4	0	Session 3
ADC Characterization	5	0	Session 3/4
Motor Characterization	4	1	Session 4/5
Functional Objective	4	0	Session 5/6
Lab Notes	3	0	After each work session, online submission
Individual Lab Note	1	0	After final lab session
Block Diagram	2	0	Session 3 onwards, online submission
Serial Decoding	0	1	Post midterm, online submission
Bug Slaying	0	0.5	Session 3 onwards, online submission

*Final project grade is capped at 30 points including bonuses.

1.4 Acceptable Collaboration and Use of Resources

For this project, you are encouraged to seek help from available resources as needed.

There is no penalty for seeking help and making use of the available resources, it is in fact encouraged. That said, it is important to get in the habit of acknowledging their use. For this course, **all resource use outside of what is provided to you on Learn must be recorded as part of your lab notes.** See Section 3 for details.

Acceptable resources and uses include:

- Internet search to find resources, investigate possible root causes, learn about tool features
- Asking a peer, TA or the instructor for help
- Using AI to find resources, terms to search elsewhere or generate ideas for possible root causes

What is not allowed:

- Providing copies of your code or any submitted work to another group
- Writing code for another group (You can provide guidance but they must do the typing)
- Asking AI to complete the project for you. You can use it for assistance, but you will be asked to explain your code as part of your demonstrations.

2 BEFORE YOUR FIRST SESSION

What you need to know before you start

The tools you will use for this project can be divided into three categories: 2-axis machine, embedded system hardware and software.

There are three different 2-axis machines to choose from this term. All include two motors and can be controlled with the provided source code using the controllers you will find in your kit.

The belt driven machine uses two parallel belts to produce movement in both the x and y directions. The machine is shown in Figure 1.

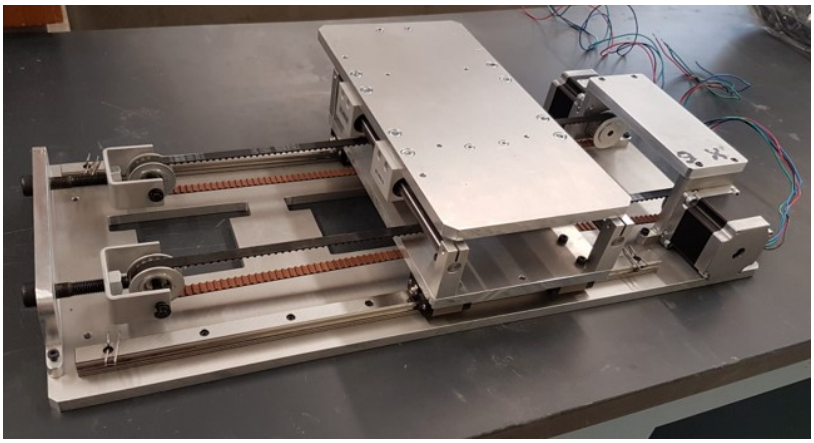

Figure 1: Pulley based 2-axis machine

The screw driven machine uses lead screws to move a platform in the x and y directions. The machine is shown in Figure 2.

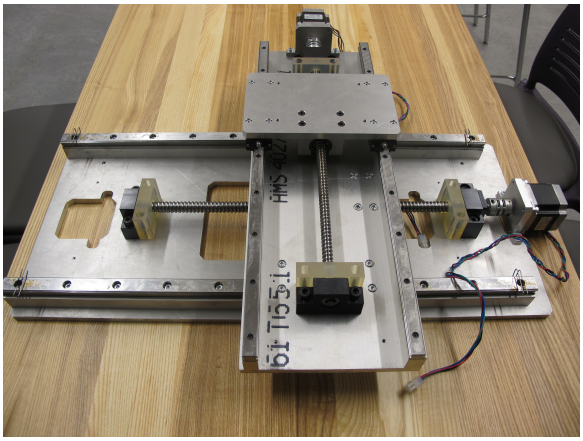

Figure 2: Lead Screw based 2-axis machine

The vertical machine uses both a belt and lead screw to move the platform up and down as well as left and right. The machine is shown in Figure 3.

Figure 3: Vertical 2-axis machine

The embedded system hardware consists of the physical components you will find in your kit. The micro-controller development board you will be using is the Nucleo F401RE as shown in Figure 4.

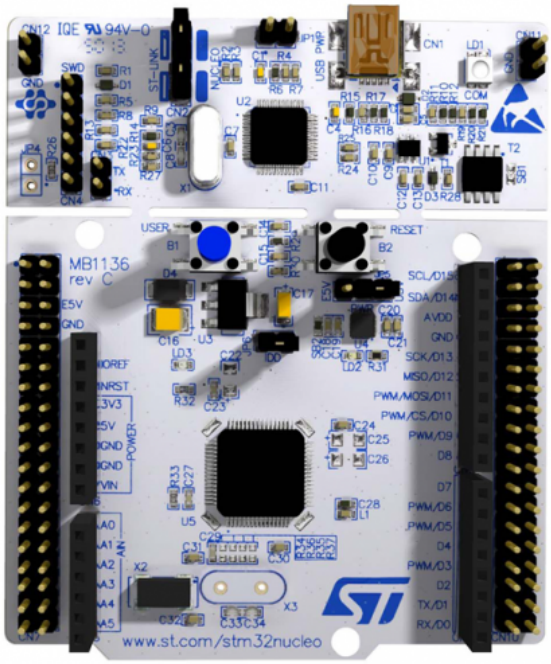

Figure 4: The Nucleo F401RE micro-controller development board

For the first lab session in this course, you will only need two components on this board: the micro-controller and the reset button. The microcontroller will start running your code when the reset button has been pressed if you are not using the debugger. The board can be connected to a computer for programming using the

micro-usb cable provided in your kit. This same connection is used to connect to a terminal window running on the computer. The board schematic and most relevant documentation can be found in the "Two Axis Machine Project -> Useful Resources" folder on Learn. More information on the development board and the micro-controller on it can be found at
http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-nucleo/nucleo-f401re.html

Attached to your microcontroller board is a motor driver shield as shown in Figure 5. This shield requires an external 8 V to 40 V supply, and is capable of independently controlling two DC motors. The board schematic and most relevant documentation can be found in the "Two Axis Machine Project -> Useful Resources" folder on Learn. For detailed information on the shield see
http://www.st.com/en/ecosystems/x-nucleo-ihm02a1.html. The sample code you are provided with includes working motor drivers. During the first lab, you should explore your ability to control motors using the spares that are not attached to the machines.

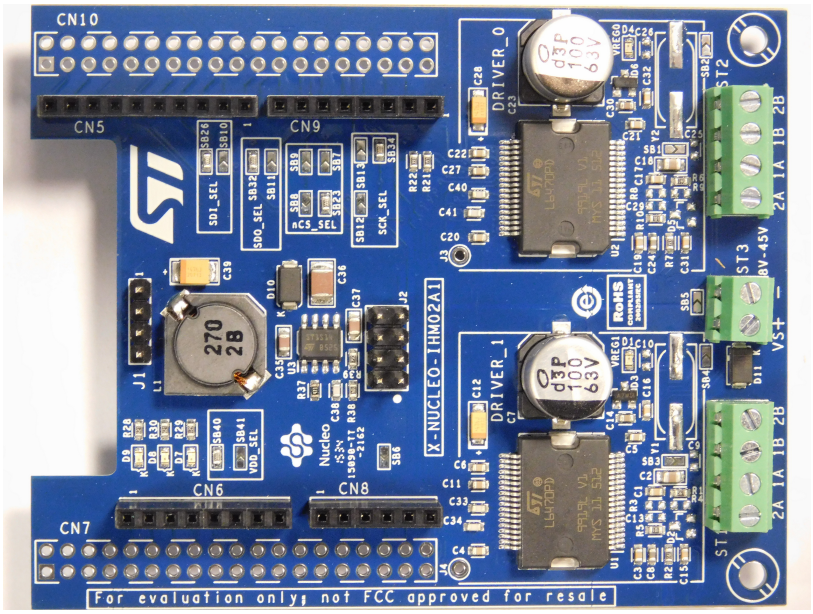

Figure 5: The Nucleo IHM02A1 motor driver shield

The second key tool you will use in this course is the software development environment, often referred to as an integrated development environment or IDE. For this course, the recommended environment is PlatformIO which runs on top of VSCode. The Nucleo board is supported by many IDEs, and you are free to use others if you like but the TAs and course instructor will not be able to support you if you have problems with the environment. The IDE is used to compile your code into a form that can be loaded and run on the micro-controller. It is also a key tool that you will use to debug your code. For reference, tips on using the PlatformIO debugger have been included in Appendix B.

Exercise 2.1: Install the IDE

Download and Install the following:

Before your first lab session, at least one group member should download and install the following on a laptop they can bring to the lab.

VS Code - This is the development environment PlatformIO runs on. It is recommended that you install with default settings if you do not already have it on your computer.
https://code.visualstudio.com/

PlatformIO - This is the extension we will be using for compilation, loading code to the board and debugging. It can be downloaded from the PlatformIO website or you can search in extensions and install from within VS Code.
https://platformio.org/platformio-ide

ST-Link Driver - If you have previously programmed ST microcontrollers from your computer, you likely have this driver already. If not, you will need to install it. For Mac/Linux users, this step should not be required.
https://www.st.com/en/development-tools/stsw-link009.html

Success Condition:
All tools in the above list have installed successfully

If you are not familiar with git, there are many great resources to get you started on the internet. One that you may find useful is
http://rogerdudler.github.io/git-guide/.

Exercise 2.2: Install Other Tools

Download and Install the following:

Git Tools - It is highly recommended you use git (or some form of version control) to manage your project code. To do so you will need to have the appropriate tools installed. Any git tools are fine, if you are not familiar with them, the following are suggested.
https://git-scm.com/downloads

Configure git credentials This step should only be necessary for a fresh install of git tools. Open a terminal window (you can use the one in VS Code or a generic terminal) and run the following commands:

git config –global user.email "*yourID*@uwaterloo.ca"
git config - -global user.name "*your name*"

Success Condition:
All tools in the above list have installed successfully

2.1 About the Source Code

The source code is hosted on the internal Waterloo GitLab server which can be accessed at https://git.uwaterloo.ca using your UW ID. There is an internally public project called MTE325_Public/W23_Two_Axis_Project you can search for or use this link https://git.uwaterloo.ca/MTE325_Public/w23_two_axis_project.

Exercise 2.3: Create a private GitLab repo (Optional)

Setup your private GIT repo

It is strongly recommended that you use version control for the project. The UW GitLab server will allow you to create a private project for your group. The instructions provided here will allow you to update your repo using VS Code integration. You are welcome to use other tools and/or methods to manage your repo as long as it is private.

- Login to the UW GitLab server and located the public project using the name or link above.
- One group member should fork the project using the `fork` button in the upper right corner. Set the visibility to `private` when you do this.
- Add the rest of the group as Project members. From the left hand tool bar, select `Manage -> Members`. Click on the blue button in the top right to invite the rest of your group.
- Each member who plans to use the git integration in VS Code will need to create an access token. This token provides access to any projects you have in GitLab, so its important that each member setup their own using their credentials. If you plan to use the command line or GUI instead of VS Code to manage your repo, you can skip to the next exercise. If you are familiar with setting up and using SSH keys, you are welcome to use them and skip the token.
  - From within GitLab, find your avatar in the upper left corner. Select `Edit Profile` from the dropdown menu.
  - Select Access Tokens from the left hand menu. Click on `Add new Token`. Give your token a name, ie. VS Code. Set an expiry date that is at least past the end of the term, and grant full access permissions.
  - Click on Create personal access token. You may wish to copy your token to notepad temporarily.

Success Condition:
You have a private repo for your group in GitLab and each member has their token required to configure VS Code.

Exercise 2.4: Install GitLab extension (Optional)

Add the GitLab extension to VS Code. Not required if you prefer to use the command line or GUI.

- From VS Code, open the extensions panel from the left menu (`Ctrl + Shift + x`) and search for the `GitLab Workflow` extension (the offical GitLab one). Install it.
- Open the Command Palette (`Ctrl + Shift + p`) then select `GitLab: Add account to VS Code`.
- When prompted, set the URL to `https://git.uwaterloo.ca` press enter, then paste the access token you created in Exercise 2.3.

Success Condition:
You have the GitLab extension installed and have linked your account.

Exercise 2.5: Setup your Project

Download and open your project in VS Code

The final step is to make a local copy of the code and test your project. The first time you open it, the required tool chains and packages automatically download and install. **This will take some time and requires an internet connection.** Please do this before your first lab.

- Clone the private repo you created in Exercise 2.3 to your local machine. A directory without spaces that is not networked and constantly syncing (ie. not OneDrive) is recommended. You can do this using GIT tools directly or follow these steps within VS Code:
  1. Click the Source Control icon in the left side tool bar then click `Clone Repository`. Select `git.uwaterloo.ca` as the source when prompted. You should now see a list of all your available projects. Select the repo you forked previously.
  2. Navigate to an appropriate destination folder with no spaces and no automatic synching. Click `Select as Repository Destination`.
  3. Select `Yes` when prompted to open the repository. You may be asked if you trust the authors. Check the check box and click `Yes, I trust the authors`.
- Go to the PlatformIO homepage tab in VS Code. If the it is not already open, select the PIO icon in the left hand tool bar then select Quick Access -> PIO Home -> Open from the menu on the left.
- Click on Open Project. Navigate to the directory with your cloned repo. You should see a `platformio.ini` file. Click Open.
- Compile your project by clicking the compile button (check mark in bottom left toolbar) or by typing `PlatformIO Build` in the command palette or using `Ctrl + Shift + B` shortcut. The build output window at the bottom should say something similar to
  `======== [SUCCESS] Took x.xx seconds ========`
  For this particular source code, you will get warnings about some typecastings and the sign of some pointers that you can safely ignore. The first time you build, VS Code may prompt you to install the C extension packs, if so, install them.

Success Condition:
You have compiled the project without errors.

Exercise 2.6: Swap Driver

Fix the driver issue in this project that results from older sample code being used with a newer version of STM drivers

While it is not something I like doing, and I certainly wouldn't encourage it in a long-term project where this change would likely be blown away by any package updates, you will need to swap one hal driver file to get the UART working. The `stm32f4xx_hal_uart.c` file in this repo will need to replace the file of the same name in your platformIO stm32 framework packages. The install location should be something similar to `C:\...\.platformio\packages\framework-stm32cubef4\drivers\STM32F4xx\_HAL\_Driver\Src`

For Mac users, the .platformIO/ folder is hidden, use `CMD + Shift + .` in the home directory to see it.

If you work with other ST boards in VS code, you may want to make a backup of the original file and replace it when you are done with the project.

Success Condition:
After you've compiled and run the sample code, you should be able to type normally in the terminal window. If you have an infinite loop of one character printing, you have not fixed your driver issue.

2.1.1 *File Structure*

Starting from an unfamiliar code base can be a challenge. The code you have been provided with has been modified from the sample projects that STM provides for their IHM02A1 motor shield. Some configuration information has been intentionally deleted. You are expected to understand the parameters and why you selected them when filling them in, but this will come after we cover analog to digital conversion. Eventually you will need to add the missing configuration information, but for now it is commented out. To help you get started on using the project, here are some notes about the code.

Before your first lab, it is recommended that you take a few minutes to explore the file organization used in the provided project. **It is not necessary, nor a good use of your time, to attempt to understand the contents of every file and function.** You will be directed to the relevant files and/or functions for each task.

`Src`	Contains the .C files for the main routine in the project as well as the motor examples. You will need to modify main and may wish to add additional source files for your custom application.
`Inc`	Contains the .h files associated with the example application.

2.2 Lab Quiz

Before working with the two-axis machines you must demonstrate familiarity with the risks and safety procedures for this lab. You will also be asked questions related to acceptable resource for this project. This deliverable is completed individually.

Tasks

- Read the SOP posted in Learn.
- Review the acceptable collaboration and lab notes sections of this document

Deliverables

- Complete the Lab Quiz in Learn. You must earn a score of 90% or above and have unlimited attempts.

3 LAB NOTES

Keeping lab notes is an essential aspect of engineering design. These notes can be critical to protecting your IP and proving timelines. Even if you are not developing IP, good lab notes are still useful. Being able to review your design decisions and the problems you encountered, as well as their solutions, can often save you considerable time as the project evolves or is shelved and revisited at a later time. In this course, good notes will help you get up and running quickly at the start of each work session. They will also help you acknowledge the resources you use.

For this course, you are required to keep lab notes. **Anytime you work on the labs/project, whether inside or outside the scheduled lab time, create a new entry** to capture any of the following that occurred during the work session:

- Which group members were involved
- Implementation decisions and details such as pin numbers, peripheral wiring, important function names and locations
- Brief summaries of any group discussions and resulting decisions
- Descriptions/details of the procedures you are following, data you collect and analysis of that data
- Notes on any problems you faced and how you solved them
- Address any specific requirements for the objective you are working on
- Acknowledge any resources used (see details below)

When you are done working for the day, submit your notes to your group's dropbox on Learn. Each entry should be appended to the same file and resubmitted. In other words, your last file will contain all notes. Earlier submission serve as timestamps and ensure all members have access to the information.

It is important to note that there is a difference between a formal report and lab notes. You should be making notes as you go, and are not expected to revise them in a formal manner. They should provide a coherent summary of the work you did, with sufficient detail that you could look back on it in the future and know why a certain decision was made or how you solved a problem you were experiencing. Spelling and grammar will not be marked, but it must be possible for someone else to read and understand what you were doing. Notes can be handwritten or typed.

All resources used must be acknowledged. Formal references are not required.

- Help from a peer: a sentence stating who the help was received from and what they assisted with
- Internet resource: copy the URL into your notes
- **AI: copy the prompt and conversation into your lab notes**

Your lab notes will be reviewed by the instructor at the end of the project and graded for completeness according to the rubric on Learn. Formative feedback on your lab notes can be requested from the instructor at any point in the term.

4 LAB SESSION 1 - DEVELOPMENT TOOLS

In lab session 1, you will familiarize yourself with the tools you need to complete this project. Please ensure you have the tools and source code you need by completing Exercises 2.1 through 2.6 before the start of your lab session. During your lab time, you will complete two exercises to prepare you for Session 2, and one exercise to prepare you for sessions 3-6. There are no marks associated with this session, but all group members are expected to attend. This session is intended to ensure everyone has the essentials working, and to get your started on the code you will need for Session 2.

4.1 Compiling and Loading Code

At this point it is assumed that you have installed all the necessary software, have created a new project and added the source code on your computer.

Compile your project by clicking the compile button (checkmark in bottom left toolbar) or by typing `PlatformIO Build` in the command palette or using `Ctrl + Shift + B` shortcut. The build output window at the bottom should say something similar to

```
================ [SUCCESS] Took x.xx seconds =================
```

From time to time, you may have warnings you can safely ignore (you should always read the warnings to be sure), but there must be 0 errors before you can load the code. For this particular source code, you will get warnings about some typecastings and the sign of some pointers that you can ignore.

> If you encounter any issues opening or compiling the project that take more than a few minutes to solve, please ask for help.

Once you have successfully compiled the code, download it to the board by pressing the `Load` button (right arrow in bottom left toolbar) or typing `PlatformIO: Upload` command palette or using the shortcut `Ctrl + Alt + U` .

To start the processor, press the black `Reset` button on the Nucleo F401RE board. At this point the code should be running but you won't be able to see anything happening.

One tool often used for debugging and program verification, or for runtime input and output from the user, is a terminal window. The terminal is a program run on the computer that the micro-controller communicates with. In our case, this is done through a COM port connection that uses the same USB cable used to program the board. The protocol used for communication between them is called Universal Asynchronous Receive and Transmit, or more commonly, UART.

UART Settings
Baud rate: 115200
Data bits: 8
Stop bits: 1
Parity: None
Flow Control: None

These settings are configured on the board side in `xnucleoihm02a1_interface.c` - `void MX_USART2_Init(void)`, but you should not change them.

In order to verify it worked, you need to connect a serial terminal to the board. To do this, you can use any serial monitor program such as PuTTY or use of the one built into PlatformIO. To open the one in PlatformIO, click the Serial Monitor button (plug symbol in bottom left toolbar) or type PlatformIO: Serial Monitor in the command palette or use the Ctrl + Alt + S shortcut. You may need to adjust the COM port number in the platformio.ini file that can be found in your project workspace by changing the `monitor_port` setting. If you are unsure how to determine which COM port the board is using ask for help. Load the code onto the development board and press the reset button. You should now see text printed to the terminal.

Preliminary experience with the VS Code monitor is that it does not play as nicely as a standalone terminal such as PuTTY. If you are having issues, try a standalone alternative.

Make sure the "echo" and "editing" options are turned on for your selected terminal program. In PuTTY, they are on the "Terminal" tab and called "Local echo" and "Local line editing". If these settings are off, you won't see the characters you type on the command line, and/or will be unable to use the backspace key.

For Mac/Linus users, you can find your COM port by typing `ls /dev/tty.*` in a terminal window. It will be something along the lines of `/dev/tty.usbmodem111403`. You will need to put this on the `monitor_port` line in platformio.ini if using the VS Code terminal.

Exercise 4.1: Tools

Check Point 1 – Working Tools:

At some point during your lab session, your group should show one of the TAs that you are able to do the following:

- Compile the provided code
- Load the code to the board
- Open and connect a terminal window (you will see text printed if you hit reset on provided project)
- Use the debugger to set a break point and check a variable (See Appendix C for tips on using the debugger)

Success Condition:
You are comfortable loading and debugging code on the board

4.2 Working with Inputs and Outputs

For the exercises in this session, you will be using General Purpose Input Output (GPIO) pins on your micro-controller. We will learn more about the inner workings of the GPIO interface when we study parallel ports, but for now let's focus on what they do. These are the ports that allow you to control individual pins on your micro-controller. These pins can also be grouped together to form a parallel interface, but for now we will control individual lines. Because of the large number of pins on the micro-controller, they are organized into groups called "ports". On the boards we are using, each port can support up to 16 I/O lines. The ports are given letter names, while the individual pins within it are numbered 0 through 15. For example, if you see "PB0" on the schematic or the data sheet, this is port B, pin 0.

The first step is to choose an available pin. To do this, you will need to **check the schematics for both the micro-controller and the shield** to find a pin that is not currently in use. For your convenience, the schematic files for both boards have been posted on Learn in the Useful Resources folder of the project. It is helpful if the pin you choose is accessible from the shield, in particular the Arduino headers are a great choice. For this micro-controller, you will need to know the port letter and pin number of the pin you wish to control.

The F401RE board actually contains two micro-controllers: one for programming and one that you are running your code on. You need to find pins connected to the one labeled MCU_LQFP64 (LQFP64 is the chip type). **Once you have selected your pin, please check the number with a TA before you connect and run your hardware.** The boards can be damaged by connecting to a pin that is already in use for another purpose.

The code you are provided with contains examples of configuring as well as reading and writing GPIO pins. In particular, you may find it helpful to review the `void MX_GPIO_Init()` function in `xnucleoihm02a1_interface.c`.

Exercise 4.2: Read from a GPIO Pin

Connect and read from a switch:

To complete this exercise, you will need to do the following:

- Find an unused pin to connect your switch to (see schematics on Learn)
- Choose how to wire your switch
- Determine the appropriate configuration settings for your pin and add them to your code. You may find it helpful to look at how other inputs in the provided code are configured.
- Initialize your selected pin.
- Read from your pin.

Success Condition:
You are able to determine the current status of your switch (open or closed)

Exercise 4.3: Write to a GPIO pin

Connect and write to an LED:

To complete this exercise, you will need to do the following:

- Find another unused pin to connect your LED to (see schematics on Learn)
- Wire your LED (if you wish to add a resistor in series to limit current, your potentiometer will work)
- Determine the appropriate configuration settings for your pin and add them to your code. You may find it helpful to look at how other outputs in the provided code are configured.
- Initialize your selected pin.
- Write to your pin.

Success Condition:
You are able to turn the LED on and off

4.3 Motor Control

The code base that has been provided for you contains working motor control code. There are two sample applications of motor control:

- `MICROSTEPPING_MOTOR_EXAMPLE` runs a fixed set of motor controls. This demo does not print anything to the console and only works if BOTH motors are connected.
- `MICROSTEPPING_MOTOR_USART_EXAMPLE` allows you to control motors individually. You can run this demo with only one motor connected, but must have a terminal window running.

Do not use the motors connected to the machines to test your code. You have not yet implemented the limit switches to prevent damage to the machines. For now, use the spare motors that will be available in the lab.

You can choose which one to run by commenting/uncommenting the appropriate `#define` lines near the top of `main.c`. To run the motors, you will also need to connect the power supply to `ST3` using the two terminal connectors provided. A setting of 12 V should be sufficient for unloaded motors. Motor commands can be found in `L6470.c`, starting around line 840. To get you started with the USART example,

commands to start and stop the motor are provided. For your project, you will have motor numbers 0 and 1. Once you have your board programmed, try typing the following in the terminal window:

Command Format	Example Command
Mx.RUN.FWD/REV.speed	`M0.RUN.FWD.1000`
Mx.MOVE.FWD/REV.num_steps	`M0.MOVE.FWD.500`
Mx.SOFTSTOP	`M1.SOFTSTOP`
Mx.HARDSTOP	`M1.HARDSTOP`

Exercise 4.4: Run the motors

Connect and run a motor:

To complete this exercise, you will need to do the following:

- Connect one of the spare motors in the lab to your driver board
- Ensure the selected sample project is `MICROSTEPPING_MOTOR_USART_EXAMPLE`
- Open and connect a terminal window
- Use the provided command samples and the full list to experiment with motor control

Success Condition:
You are able to start, stop, change the direction and change the speed of the motors.

Going Further If you have additional time in your lab session, start to think about how you could extend the reading of a GPIO to polling and interrupts.

5 LAB SESSION 2 - SYNCHRONIZATION

In lab session 2, you will complete exercises on polling and interrupts to explore their use as synchronization techniques. For the exercises in this session, you will need to use an oscilloscope and signal generator in addition to the hardware in your kit. You will also need a USB stick to save scope images.

5.1 Background Information

You may wish to refresh your knowledge of working with the oscilloscope and signal generator. You should be comfortable connecting probes, setting up your scales and adjusting the trigger source.

One additional feature you will need to use on the scope is persistence. This feature allows you to easily compare repeated samples by overlaying them. On the Keysight oscilloscopes, it is enabled by pressing the Display button then selecting persistence from the on-screen menu.

You will build on the GPIOs you added last session to test both polling and interrupts. For this micro-controller, the interrupt vector names are fixed. You can find the full list of available interrupts in `startup_stm32f401xe.s`. This file declares infinite loops for all interrupts as "weak" functions. This means unless you declare your own handler with the same name, the default is called and your code will hang in the infinite loop. In the project code provided, the ISRs are contained in `stm32f4xx_it.c` and you can refer to these examples when implementing your own. To avoid issues in future labs, it is recommended that you make use of an unused ISR and do not modify those that are currently in use. You may also find the following documentation useful for selecting pins and configuring interrupts:

- STM32F401 Reference Manual (RM0368) pages 201-211
- STM32F401 Programming Manual (PM0241) pages 36-43

Links to both documents are located on the STM32F401RE homepage,
http://www.st.com/en/microcontrollers/stm32f401re.html

5.2 Results Summary

For this session, you will need to document and analyze the results of your experiments. Each exercise contains details on what must be included in your summary. No title page is required and your submission should be a maximum of 6 pages, including all required figures. The required code samples should be copied at the end of your report, but are not included in the page count. This written summary must be submitted to the Dropbox on Learn by 8:00 pm one week after your lab session.

5.3 Hardware Setup

For exercises 1 and 2, you will need to use the hardware in your kit as well as the oscilloscope and signal generator in the lab. Connect the following:

- Connect the signal generator as a GPIO input
- Configure your signal generator to produce a square wave with a 3.3 Vpp voltage at 100 Hz
- Connect your LED from session 1
- Connect one input of the oscilloscope to the output of the signal generator and the other to the LED on the pin side

Exercise 5.1: Tight Polling Latency

Objective: Find the maximum frequency at which you are able to reliably respond to a rising edge from the signal generator using tight polling.

To complete this exercise, you will need to write code to do the following:

- Tight poll for a rising edge from the signal generator
- Turn on the LED in response to the rising edge, and ensure it is turned off before the next rising edge
- Respond to the rising edge of successive pulses as fast as possible

Success Condition: You are able to see both the square wave from the signal generator and your system's response to it on the oscilloscope.

Results Summary: What is the maximum frequency square wave which you are reliably able to respond to using polling? Include the following in your response:

- Document the procedure you used to find the maximum frequency.
- Use a USB stick to capture a screen shot showing the maximum frequency and your response with persistence enabled for at least 100 ms.
- How did you determine what a reliable response was?
- What is the variation in your response time? Hint: use persistence and cursors for measurements
- What might be causing the variation in response time?
- Include a copy the code used for polling at the end of your report

The scopes are capable of saving many file types. Make sure your images have been saved as image files before you proceed. If the file format is incorrect and you are unsure where to adjust the settings, ask a TA for help.

Exercise 5.2: Interrupt Latency

Objective: Find the maximum frequency at which you are able to reliably respond to a rising edge from the signal generator using interrupts.

To complete this exercise, you will need to write code to do the following:

- Configure the input pin connected to signal generator to use interrupts
- Implement an interrupt handler to turn on the LED output in response to a rising edge from the signal generator. Make sure it is turned off before the next rising edge.

Success Condition: The interrupt handler is called for each rising edge on the signal generator input, and you are able to see the signal and the response on the oscilloscope.

Results Summary: What is the maximum frequency square wave which you are reliably able to respond to using interrupts? Include the following in your response:

- Document the procedure you used to find the maximum frequency.
- Use a USB stick to capture a screen shot showing the maximum frequency and your response with persistence enabled for at least 100 ms.
- How did you determine what a reliable response was?
- What is the variation in your response time?
- What might be causing the variation in response time?
- Include a copy of all functions called when the interrupt is handled at the end of your report.

Exercise 5.3: Analysis

To complete this exercise, you will need your results from exercises 1 and 2. Answer all of the following questions using your knowledge of course concepts as well as experimental results.

Results Summary:

- Draw a schematic diagram of your system which includes the port and pin numbers of your input and output connections used for the exercises, as well as the oscilloscope connections. You do not need to include any other pins on the micro-controller or other unused components on the board/shield. Enough detail should be included for someone to connect the hardware required for the experiments without any additional information. You may hand draw the diagram as long as it is tidy and legible. To draw it electronically, you may find a tool such as Visio or SchemeIt (free) useful.
- Which technique resulted in a lower latency between the rising edge of the input and the response? Why?
- The choice between polling and interrupts is often a matter of trade-offs.
  - What are two benefits of polling? Briefly explain.
  - What are two benefits of interrupts? Briefly explain.
  - Which synchronization technique do you think will be more appropriate for the limit switches on your two-axis machine? Justify your choice.
  - What could go wrong? What impact would this have and how can you prevent it?

5.4 Marking Rubric

The work completed in this session is worth 6 points out of 30 on your project. Your results summary submission will be graded in Learn using the rubric below.

Synchronization Techniques (24 marks)

Implementation (4 marks)

	4 marks	3 marks	2 marks	0 marks
Polling and interrupts are implemented	Both synchronization techniques are working. The interrupt handler follows best practices and tight polling is properly implemented.	Both techniques are working but improvements could be made to interrupt handling OR tight polling implementation.	Significant issues are present in the implementation of one or both techniques.	Not demonstrated

Knowledge (16 marks)

	4 marks	3 marks	2 marks	0 marks
All questions from Exercise 1 answered. Explanation and insights provided to show understanding of polling beyond observations.	Consistently	Mostly	Partially	Not demonstrated
All questions from Exercise 2 answered. Explanation and insights provided to show understanding of interrupts beyond observations.	Consistently	Mostly	Partially	Not demonstrated
All questions from Exercise 3 answered. Explanation and insights provided to show understanding of trade-offs between synchronization techniques.	Consistently	Mostly	Partially	Not demonstrated
The schematic is complete and tidy. It can be used to connect the necessary hardware.	Consistently	Mostly	Partially	Not demonstrated

Quality (4 mark)

	4 marks	3 marks	2 marks	0 marks
Report quality is consistent with a 3A level student. Reports will be compared with peers.	Report is organized in a logical manner with only minor spelling and/or grammar mistakes.	Report is generally well organized but improvements could be made or there are multiple spelling and/or grammar mistakes	Report is disorganized and/or there are many spelling and/or grammar mistakes.	Not submitted

6 LAB SESSION 3-6

The remaining lab sessions are less procedural than what you have seen so far in sessions 1 and 2. You have reached the phase of the project where you have the building blocks you need to start making design and implementation decisions that will affect the behaviour of your machine. You should use the next four lab sessions to complete all of the objectives described here, some of which are demonstration based and some of which you will submit on Learn. The demonstrations should be done as each objective is completed and typically last 5-10 minutes. To use you scheduled lab time as effectively as possible, you should be coming to each lab session with code ready to test on the machine. It is expected that you will spend an equal amount of time on the project outside the lab as you do in lab sessions (roughly 18 hours for the term).

Demonstrations:

- Limit Switch Implementation
- ADC Characterization
- Motor Control
- Functional Demonstration

Online Submissions:

- Block Diagram
- Group Lab Notes (submit after each **work** session)
- Final Lab Note (after session 6)
- Bug Slaying (optional bonus)
- Serial Decoding (optional bonus)

6.1 Demonstration Based Objectives

The Demonstration Marking Sheet is available on Learn, please print a copy for your group and have it available for the demonstrations. Hand it in to the instructor after the last demonstration is marked.

6.1.1 *Objective D1: Limit Switches*

Safety first! Before you run the motors on your two axis machine, ensure they will stop when the plate reaches the end of its travel range.

Tasks
Modify the code to add the necessary interface(s) to support 2 limit switches on each axis (4 switches total). Add functions to stop or reverse the motor when the limit switch is tripped. What are the edge cases? Write tests for these cases.

Tip: You can (and should) test this with a motor that is not connected to the machine. Motors that are not mounted to machines and power supplies will be available in the lab for you to use during your scheduled sessions.The switches mounted to the machine can be used for development and testing of this objective.The motors on the machine should not be run until you have demonstrated this objective and a TA has given you permission to proceed.

Deliverables

- Document your test cases – what the test set up is, what the test set up tests in the code, and results for each test. Also document actions taken if the experiment fails or partially fails.
- Demonstrate your code. Be prepared to explain your implementation choices and how you tested it.
- Demonstrate that the limit switches prevent the platform from hitting the crash blocks or end of the rails on the 2-axis machine.

The platform must be able to travel safely in both directions with both motors running. To earn full points, the platform must be able to reach the end of both rails safely and stop or reverse as soon as a limit switch is triggered. Proper corner case handling must also be demonstrated before you will be allowed to being running the machines to complete the remaining demonstration objectives.

6.1.2 *Objective D2: ADC Characterization*

Use a potentiometer to test and characterize the ADC functionality.

1. Connect a potentiometer to the ADC of the NUCLEO Board.
2. Verify your ADC is working and able to detect input changes.
3. Design and document experiments to characterize the behaviour of your potentiometer and ADC.
4. Perform your experiments and document your results.
5. Reflect on your results by answering the following questions:
   a) Are the results reasonable?
   b) Were any problems with the experimental design found when conducting the experiment?
   c) How would you change your procedure to improve your results on a future run?
6. Repeating your experiment is not required unless you determine that your results are not reasonable or more data is needed. If you do repeat it, document your changes and the results.

Hint
You will need to configure your ADC settings first by filling in the settings for the struct that is populated using the `void MX_ADC1_Init(void)` function in `xnucleoihm02a1_interface.c`. Your ADC should work in polling mode at this point. Should you choose to use interrupts (optional), some additional tweaks to existing functions may be needed.

In order to help you get started, some sub-tasks have been listed below with associated questions you should address and document in your lab notes. This is not an exhaustive list of sub-tasks, and not all listed sub-tasks necessarily need be done, depending on your implementation decisions. They are supplied as a starting point to consider. The tasks are not necessarily dependent on one another. It is up to you to decide if they need to be completed, what order to complete them in, and where dependencies exist. Add additional tasks as you see fit.

SUB-TASK	LAB NOTES
ACTIVATE YOUR INTERNAL ADC	• What is required to turn it on? • Modify your code to do so
CONNECT YOUR POTENTIOMETER	• Design and build your potentiometer circuit • Check the data sheet to determine which pins can be used as inputs to the ADC and the corresponding channel • Connect it to the Nucleo board • Modify your code to support this connection
CONFIGURE YOUR INTERNAL ADC	• What resolution did you choose? Why? • What conversion mode? Why? • What is your sampling time? Why? • What happens when you change clock settings?
TEST YOUR ADC	• How can you ensure your ADC is enabled and working?
CALIBRATE YOUR ADC	• Ideally your ADC is linear. How accurate is it? • What is the error in your ADC?
CALIBRATE YOUR POTENTIOMETER	• Define and carry out a calibration procedure • Does it behave linearly? • How can you map the ADC readings to motor speed?

Deliverables

The following should be included in your lab notes, and will be reviewed as part of the demonstration:

1. Characterization procedures and results (properly formatted) for both the potentiometer and ADC
2. Results of conducting your experiment
3. Reflection on your experiment results

You will need to demonstrate your ADC in action. Be prepared to discuss your implementation decisions and share your procedures and results with your marker. See Demonstration Marking Sheet for grading details.

Hint

"Because we found sample code on the internet with these settings and it works" is not a sufficient explanation of your settings choices.

6.1.3 *Objective D3: Motor Characterization*

Explore the behaviour of your motors and alternative control options. Recall that the code you are provided with is capable of controlling the motors from the command line, as was tested in the first lab session. Now it's time to explore other available commands and the motor behaviour.

Tasks
Once your limit switches and ADC work, determine an appropriate range of motor speeds and movement patterns. Things you should consider include:

- What control commands are available to you? How do they work?
- Mapping of potentiometer position to ADC values to motor control
- Appropriate speed or distance settings (What is your min? What is your max?)
- How can you safely handle direction changes?
- Edge cases of operation and how you will test them

Document your discussions on the above in your lab notes. You may be asked to present your notes or discuss them during the demonstration.

Hint
Testing is simpler if you break systems into smaller pieces. Being able to control the motors without the command line before you try to integrate control with the ADC will make things easier.

Deliverables

- Document your edge cases – what the test set up is, what the test set up tests or demonstrates, and results for any tests you ran.
- Demonstrate your code. Be prepared to explain your implementation choices and how you tested them.
- Explain the min and max useable settings. How were they found?
- Run your code and show the machine in action with at least one motor being controlled by a potentiometer.

See Demonstration Marking Sheet for grading details.

Hint
You may find the functions in `L6470.h` helpful

Bonus
Bonus points can be earned by connecting two potentiometers and controlling both motors at the same time. For full points, motors should be responsive to both potentiometers without any significant lag when both moved at the same time.

6.1.4 *Objective D4: Functional Demo*

The completion of the demonstration and submission based objectives will help your group work towards the functional requirement for this term. Scoring for the functional objective will take place during session 5 and 6.

Winter 25 Functional Objective: Precisely position your platform to align with a target. Scoring will be based on three categories:

- Precision: How well aligned you are to the hanging target, scored using a bullseye that will be placed on the platform
- Repeatability: The time to reach each position will be measured 3 times, scored on consistency of time measurements
- Control: Demonstrate the ability to maneuver the platform safely. All limit switches must still be functional for full marks.

6.2 Submission-Based Objectives

The following may be completed at any time before the project deadline. Rubrics for these objectives are on Learn and should be accessible through the respective dropbox on Learn.

6.2.1 *Objective S1: Individual Final Lab Note Entry*

Once you have completed all objectives, each team member must write an individual final entry. This final entry will be submitted to an individual drop box. This entry must be no more than a page and answer the following questions:

- What was the most challenging part of the project for you?
- What part or parts of the project contributed the most to your learning?
- What would you do differently if you were to complete another embedded systems project in the future?
- What wouldn't you change if you were to complete a different embedded systems project in the future?

Deliverables
Submit your work to the "Final Lab Note (Individual)" dropbox on Learn. The grading rubric is attached to the dropbox.

6.2.2 *Objective S2: Block Diagram*

Document your system architecture

Tasks
Draw a block diagram of your embedded system implementation. The purpose of the block diagram is to provide a tool that would allow someone not familiar with your project to connect the hardware as required, and to have a high level understanding of the relationship between components. It should also have sufficient information to allow debugging your system without having to consult other documentation. It must be done electronically. You may find a tool such as Eagle, SchemeIt or Visio helpful. It must include the following:

1. Blocks for all major system **components** (A box for the dev board and a box for the shield is not specific enough. Thinks about the chip level.)
2. Pin numbers for connections between the micro-controller and external components
3. Signal names for external connections, use protocol specific names where appropriate
4. Label protocols (ie. SPI, I2C, USART) where appropriate

Deliverables
Upload your final diagram to the "Block Diagram" dropbox on Learn. The grading rubric is attached to the dropbox.

6.3 Bonus Objective

6.3.1 *Objective B1: Bug Slaying*

In approximately one page, describe the following:

- A particularly difficult bug your team encountered that you spent at least 2 hours trying to resolve (root cause may be hardware or software)
- How long you spent on the problem
- The tools/strategies you used to debug it
- Two things you tried that didn't work
- How you eventually identified and resolved the root cause
- What you learned from the experience

Deliverables
Submit your work to the "Bug Slaying" dropbox on Learn. The grading rubric is attached to the dropbox.

6.3.2 *Objective B2: Serial Decoding*

Capture and decode samples of two different serial protocols in your system.

Tasks
Use an oscilloscope to capture a trace of each of your selected serial transaction protocols.

- Identify the pins used for the serial protocol and connect scope probes to them. Please use jumper wires between the probes and the board as directly connecting the probes has short circuited boards on the past.
- Use the scope with appropriate trigger, horizontal and vertical resolution settings to capture a sample where each bit can be identified. A complete frame must be captured in each image.
- Use a USB key to save an image of your captured sample. Mark up your scope captures by hand or electronically. Your marked-up version should be accompanied by a short write up that includes:
  - On your image, label the signal names, start and end of each frame, and the sampling point for each bit.
  - Decode the bits. Write out the binary values. Identify which bits are data. Label any remaining bits and briefly explain their purpose.
  - Indicate whether the data bits represent raw data. If they do not (ie. are ASCII), indicate the final decoded value.

Deliverables
Submit your work to the "Serial Decoding" dropbox on Learn. The grading rubric is attached to the dropbox.

Hint
You do not need to write any code to complete this deliverable. There is traffic on at lest two serial buses if you have the terminal and the motors running. This can be achieved using the provided code. No hardware modifications are required for this task, but there are some critical details on the schematics that can be easy to miss. Check them carefully. You may also find that the configuration information in `xnucleoihm02a1_interface.c` is useful.

A DEBUGGING PRINCIPLES

The following provides a review of the principles of debugging. This material was originally developed for use in MTE 241, but has been included here for review purposes.

a.1 About Debugging

Most IDEs provides a variety of debugging tools. How can we use them and why are they essential to programming efficiently for an embedded system? Why do we need multiple views within a tool? To answer these types of questions, first we need to understand what debugging is.

Practitioners and researchers do not agree on a unique definition of debugging. Many terms, such as program testing, diagnosis, tracing, or profiling, are interchangeably interpreted as debugging or as part of a debugging procedure. Researchers define debugging as an activity requiring localization, understating, and correction of faults [1]. Therefore, to debug code, the first step is to localize the fault. In a large software project with millions lines of code and many source files, it would be very hard or even impossible to check them all manually. We need tools to help us. Modular programming is also helpful here, because we can easily review a limited number of modules related to the fault.

A lack of understating the root cause of the fault can result in a fix that only corrects the symptoms, not the actual problem. Therefore, once the part of the code causing the fault is localized, it is essential to figure out the actual reason or source. Fixing the fault is the last step of debugging procedure. This step also includes verification to ensure fault is mitigated and no new issues have been introduced.

a.2 Testing

Debugging is started once a fault is revealed. One way to reveal and model faults is software testing. A test-case is an input and output pair that demonstrates the expected behaviour of the software. If the actual output differs from the expected one, a fault has occurred. A test-case, as a scenario of an execution, can be functional or non-functional.

Functional test-cases check whether the output matches what is expected by the user. For example, a program computing the square root is expected to return 2 when the input is 4.

Non-functional test-cases examine the quality of a given software system. Unlike most non-functional properties, performance is an important non-functional property that can easily be tested. Measuring the elapsed time, the number of finishing tasks in a unit of time, and the number of concurrent clients are some candidates for testing performance in any software system. Recall that real-time systems have a strict

restriction on time performance: a particular number of tasks have to be done in a certain amount of time.

It is essential that a test-case be reproducible. A failed test-case, as a specification of a fault, has to result in the same output no matter how many times it is executed. Making reproducible test-cases for concurrent software systems can be very hard and may be impossible. So, one may need to simulate the same situation within a test-case.

a.3 Functional Debugging

Given a test-case revealing a fault in functionality of the software, one has to first localize the debugging area and try to understand the potential causes. Debugging a fault can be done using static or dynamic information. Static debugging is performed using the code without considering any execution traces. As an example, when one debugs a fault in the multiplication operator of a calculator program, they only look at the modules doing the actual multiplication calculation and the rest can be safely ignored. The complexity of using programming structures, such as loop or if statements, makes static diagnosis really hard or impossible in some cases. To address this, debugging makes use of dynamic traces of the program's execution. Any test-case represents one execution trace of a given program, whereas the code executing the test-case may be capable of producing more than one trace. Once a faulty trace is found, the localization and understating becomes focused on the code generating that trace.

There are several techniques used to find the appropriate execution traces. Some require instrumenting the code (adding additional code specifically for debugging purposes) and others use the debugging tools. Some techniques are useful for basic faults and others for more complex ones. In the course of debugging any non-trivial program, it is likely that a combination of techniques will be used. Three common debugging techniques are described below.

a.3.1 *Tracing the Code Step-by-Step*

Debuggers provide facilities for executing the code line-by-line. After executing each statement, one may see the contents of the stack, memory locations, registers, variables, and ports and check how the executed statement affects them. Single step tracing gives very detailed information, but becomes cumbersome or infeasible debugging repetitive and/or long traces. It is most useful when a suspect cause can be isolated to a small portion of the code, in which case stepping through the code can be used in conjunction with a breakpoint.

a.3.2 *Breakpoints*

Instead of stepping into every statement, one can use breakpoints to stop the execution on particular lines. Once the program execution is stopped at a line, all the execution information becomes visible for checking or even changing. Breakpoints are very powerful debugging tool that are used to stop the execution on specific statement or specific condition. For example, one might set a conditional breakpoint to pause the execution if a variable is read or written to. One might also set a breakpoint at the first line of a function to be verified or suspected as the root cause of a fault.

a.3.3 *Instrumenting with `printf`*

Using `printf` statements is a common and effective debugging technique that is used by programmers [2]. For debugging a fault, a programmer instruments the code by placing `printf` statement in particular locations to see how variables are changed during test-case execution. Debugging with `printf` statements only requires a compiler and does not require an additional debugging tool. The problem with using `printf` statements in real-time systems is that the `printf` command may not be always available. Moreover, instrumenting the code with `printf` statements is not repetitive and some statements have to be changed from one test-case to another one. Adding additional `printf` statements requires the code to be recompiled, and can require extensive code cleanup once issues are resolved. They also consume a large amount of resources, both CPU cycles and memory. Therefore it is helpful to have alternate debugging tools available.

B DEBUGGER

b.1 Using the Debugger in VS Code

After successfully compiling your code, click the Run and Debug icon on the left hand side (Play button with a bug) or select `PlatformIO: Start Debugging` from the command palette or use `Ctrl + Shift + D`.

When the debugger starts, you should see something similar to Figure 6 depending on the view configuration. Press F5 or the green Play button to start the session.

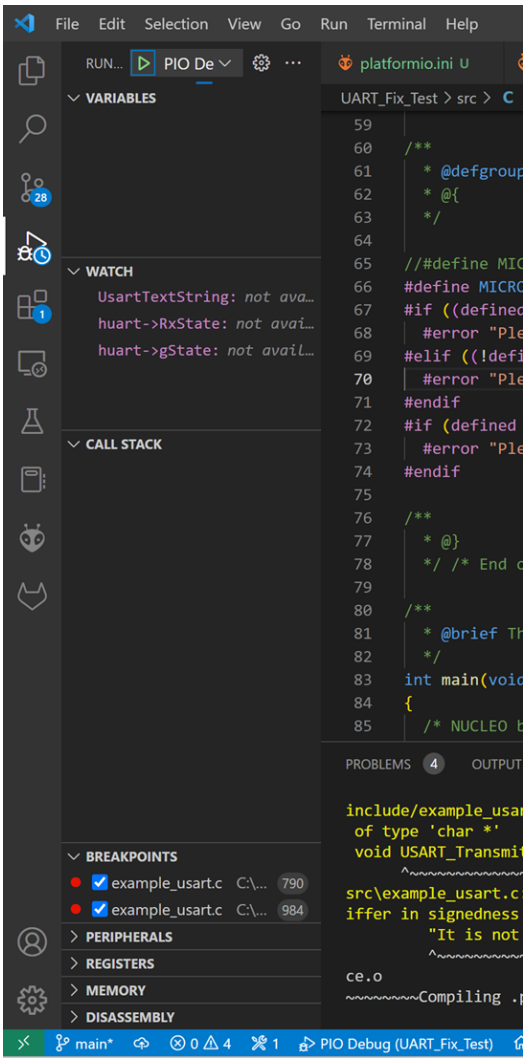

Figure 6: The debug view. Yours may appear slightly different depending on which windows are enabled and how they are arranged.

The debug controls are shown in Figure 7. They allow you to start the program, stop the program and step through the code in a couple different ways. Holding your

mouse over each of the icons will show a tool tip explaining the functionality of the button as well as its associated Fn shortcut key.

Figure 7: Debug Control Toolbar

For Project 1, we will introduce hardware break points, and the Call Stack Window, Watch Windows and Memory Windows.

> If you find the variable you wish to look at doesn't exist in the debugger it has likely been optimized out. You can confirm this by adding the `volatile` keyword in front of the variable declaration, recompiling, then restarting the debugger.

b.2 Hardware Break Points

Being able to stop the code at a desired point in order to "see" what is going on is one of the primary uses of a debugger. This is done using break points.

A breakpoint can be placed at any executable line of Assembly or C code. To place a breakpoint, find the line you want to stop on, then click just to the left of the line number. A red circle on the left side of the intended code shows the breakpoint has been placed and the execution will stop here when the program is run. Breakpoints can be disabled or removed via the `BREAKPOINTS` pane in the debug view or by clicking the red circle next to the line number.

Once the execution is paused on a statement, the processing unit is stopped and the program counter does not proceed to the next statement, so one can see the call stack content, registers' values, watched variables, and port values. Like any other debugger, you can use the Step Into, Step Over and Step Out from the next statement buttons, or Run To Cursor Line to advance the program. The execution trace can also be continued using the Run command, or terminated using Stop command.

b.2.1 *Call Stack and Variable Pane*

The `CALL STACK` and `VARIABLES` panes shows you two things: the sequence of function calls that have been made, and the status of the variables that are currently in scope. An example is shown in Figure 8.

b.2.2 *Watch Pane*

Using the `WATCH` pane, one can see the content of any variable at any time while the execution is stopped. To watch a variable, click on the plus symbol next to WATCH

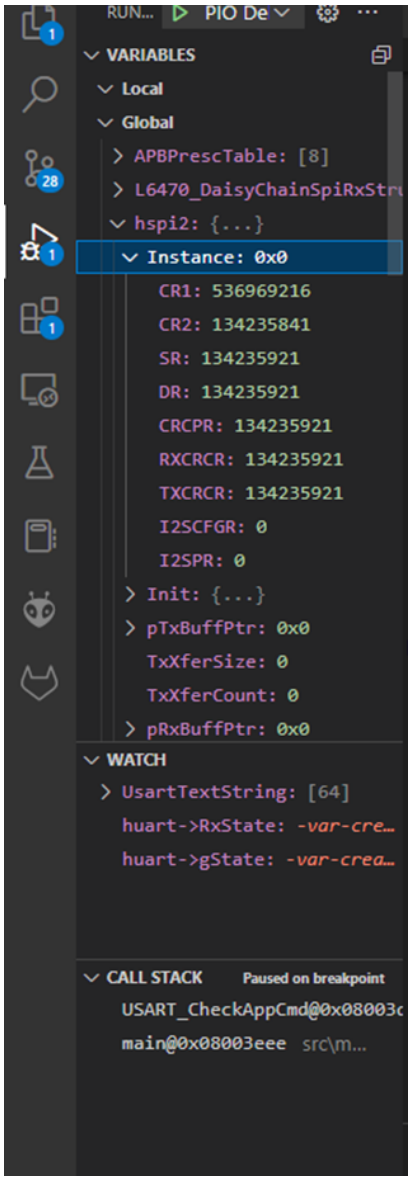

Figure 8: Sample view of the call stack and local window

then type the variable name. The content of the variable becomes visible whenever the variable is in scope within the current trace.

b.2.3 *Memory Pane*

The MEMORY pane allows you to inspect the current contents of memory while program execution is stopped. Enter a starting address then the number of bytes you wish to display.

Acknowledgements

Parts of the material shown here were adapted from material originally prepared by Douglas Harder for MTE 241. Adaptations and additions have been made by Allyson

Giannikouris for MTE 241, and further updates have been made for VS Code and MTE 325.

REFERENCES

[1] J. W. Valvano. *Embedded systems: Introduction to ARM Cortex-M microcontrollers*. self published, 5th edition, 2014.

[2] J. W. Valvano. *Embedded systems: Real-time operating systems for the ARM® cortex-M microcontrollers*. self published, 2nd edition, 2014.