MTE 325 - PROJECT 1

ALLYSON GIANNIKOURIS*, P.ENG.

Spring 2020

Version 1.0

CONTENTS

1 INTRODUCTION

The projects in this course 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.

Each student will complete their own project and submissions. That said, you are encouraged to talk to one another and the teaching team when you run into issues. Being a sounding board and suggesting possible issues to each other is fine. Sharing code or writing code for another student is not permitted.

1.1 Learning Objectives

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

- Use industry grade tools and equipment to configure and debug an embedded system
- Configure and use parallel ports in an embedded system
- Implement a tight polling loop

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

2 GETTING STARTED

What you need to know before you begin

The tools you will use for this project can be divided into two categories: embedded system hardware and software.

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 Discovery 32L476 as shown in Figure 1.

Figure 1: The Discovery L476 micro-controller development board

The micro-controller 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 mini-usb cable. 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 "Project -> Useful Resources" folder on Learn. More information on the development board and the micro-controller on it can be found at

www.st.com/en/evaluation-tools/32l476gdiscovery.html

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 Keil μVision 5 (Windows) or System Workbench (mac/linux). The Discovery 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 Keil

debugger have been included in Appendix D and similar tips for System Workbench can be found in Appendix **??**.

Exercise 2.1: Install the IDE - Windows

Download and Install the following:

Install the tools on the computer you will use for projects.

uVision 5 IDE - This is the recommended IDE for the project if you are running Windows. A project file for this IDE has been provided, and this is the tool the teaching team is familiar with. It is recommended that you use the default settings and paths for installation. When installation completes, the pack installer may open automatically. If it does, close it then close uVision 5. We will get to configuring the IDE after you have the project code ready. You may use another compatible IDE, but this is only recommended if you have experience setting up embedded systems projects.
http://www2.keil.com/mdk5/install

ST-Link Drivers - These are the drivers for the programmer that is integrated on the Nucleo development board. They need to be installed for the computer to recognize that a board is connected. Scroll down to the "Tools and Software Section". Select the latest driver, which at the time of this writing is "STSW-LINK009" to download and install.
http://www.st.com/en/development-tools/st-link-v2.html

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

Exercise 2.2: Install the IDE - Mac/Linux

Download and Install the following:

Install the tools on the computer you will use for projects.

System Workbench IDE - This is the recommended IDE for the project if you are running MacOS or Linux only. A project file for this IDE has been provided, and the teaching team has tested it on macOS. It is recommended that you use the default settings and paths for installation. the install is quite large and will take some time. When prompted to install drivers, install them. You may use another compatible IDE, but this is only recommended if you have experience setting up embedded systems projects.
www.openstm32.org

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

Exercise 2.3: 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

Terminal Program - There are many free terminal programs available. If you do not already have one installed and are not familiar with them, try downloading any one of the following: PuTTY, Tera Term, or Real Term.

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. It is strongly recommended that you create a new **private** repo to store all your projects this term. 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 2.4: Access the Code Base

Setup your private GIT repo and checkout the code

It is strongly recommended that you use version control for the project. The UW GitLab server (`git.uwaterloo.ca`) will allow you to create a private project. You are welcome to use other version control tools as long as the repo is private. You should be able to copy the code from the provided course repo. `https://git.uwaterloo.ca/MTE325_Public/s20_project`

Success Condition:
You have a private repo with the provided code in it.

Exercise 2.5: Load Support Packages (uVision 5 only)

Load/update the necessary packages in uVision

Now that you have the project code, it is recommended that you load the necessary support packages in Keil uVision prior to your lab. These downloads can take some time.

At this point it is assumed that you have installed all the necessary software and have checked out a copy of the source code on your computer. Navigate to the folder below to find the project file.

`check_out_loc\Project1\MDK-ARM`

Double click on "`Project.uvprojx`". Install the support packages using the instructions in Appendix A.

Success Condition:
The packages described in Appendix A show they are installed.

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. The code should not compile on checkout. Eventually you will need to add the missing configuration information, but for now comment it 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. The code has been divided into two main folders: Drivers and Project code. **It is not necessary, nor a good use of your time, to attempt to understand the contents of every file.** You will be directed to the relevant files for each task.

`\Drivers\`

`STM32L4xx_HAL_Driver`	Hardware Abstraction Layer (HAL) drivers. These are standard files for the STM32 family. **These files should not be modified.**
`CMSIS`	Cortex Micro-controller Software Interface Standard (CMSIS) drivers. These are the standard drivers for the Cortex processor core. The README.txt file in this folder has more details on what is in each folder. **These files should not be modified.**
`BSP`	Board Support Package (BSP) drivers. These files contain code that is specific to the configuration of the Discovery L476 board. There are files for each of the peripherals that can be initialized.

`\Project1\`

`MDK-ARM`	Contains the uVision 5 project file, as well as other files used by the IDE.
`SW4STM32`	Contains the System Workbench project file, as well as other files used by the IDE.
`Src`	Contains the .c files for the main routine in the project. 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.

3 PROJECT 1 - DEVELOPMENT TOOLS

In project 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 2.4 before you start. This project is intended to ensure everyone has the essentials working, and to get your started on the code you will need for Project 2.

3.1 Report

For this project you will complete and submit an informal report to a dropbox on Learn. A Word template is available on Learn. You are encouraged to discuss issues with each other and post questions on Piazza. Helping each other with ideas is fine, but you must not directly provide or copy code. Directly copying report responses is also not allowed.

The rubric that will be used for grading is available on Learn.

3.2 Compiling and Loading Code

At this point it is assumed that you have installed all the necessary software and have checked out a copy of the source code on your computer. If you are using System Workbench as you IDE, refer to Appendix B instead.

Navigate to the folder below to find the project file. `check_out_loc\Project1\MDK-ARM`

Double click on "`Project.uvprojx`". If this is the first time you are opening a project, you will need to install the support packages using the instructions in Appendix A. If this is not the first time, it should open without errors. If there are any warnings about missing files, please double check that you installed all required packages.

Compile your project by clicking the compile button, as shown in Figure 2. The build output window at the bottom should say something similar to the line below, where `Project` is your project name

```
"Project.axf" - 0 Error(s), 0 Warning(s)
```

if everything was successful. 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.

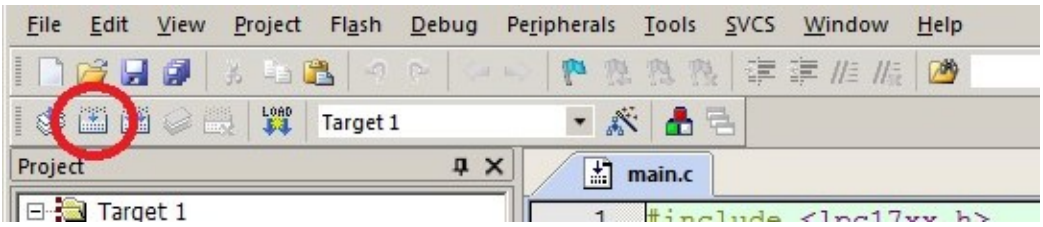

Figure 2: Location of the `compile` button on the tool bar

> 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 as shown in Figure 3.

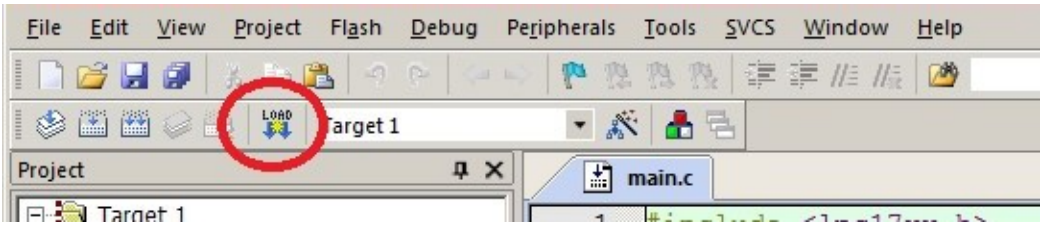

Figure 3: Location of the `load` button on the tool bar

To start the processor, press the black `reset` button on the Discovery board. At this point the code should be running but you won't be able to see anything happening. In order to verify it worked, you need to press straight down on the blue joystick. The red LED should turn on. Press the joystick again and the LED should turn off.

Exercise 3.1: Tools

Check Point 1 – Working Tools:

Before you write any new code for this project, you should confirm your tools are working by doing the following:

- Compile the provided code
- Load the code to the board
- 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

3.3 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 Discovery board we are using actually contains three micro-controllers: one for programming, one that you are running your code on, and one that is used as an onboard ammeter for current measurements. Take some time to look through the schematics. It may not all make sense right now, and that's ok. For now, identify the STM32L476 as well as the red and green LEDs that are user IO and the joystick connections.

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 how `void BSP_LED_Init()` function in `stm32l476g_discovery.c` is used to configure the red led.

Exercise 3.2: Write to a GPIO pin

Connect and write to an LED:

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

- Use the schematics to find the port and pun associated with the green LED
- Determine the appropriate configuration call and add it to the global initialization in main. You may find it helpful to look at how other LEDs in the provided code are configured.
- Toggle your LED state once per second. Do not write your own toggle function, use what is provided.
- Copy your code into your Project 1 report.

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

Exercise 3.3: Read from a GPIO Pin

Initialize and read from a button:

The blue joystick on the board is nothing more than 5 buttons packaged together: up, down, left, right and select. To complete this exercise, you will need to do the following:

- Use the schematics to find the port and pin associated with "down" on the joystick
- Search the part number "MT-008A" to determine how the joystick works internally
- Determine the appropriate configuration settings for your pin and add them to your code. You may find it helpful to look at how the select was configured using `void EXTI0_IRQHandler_Config(void)` in `main.c`. There is code here you won't need as it is used for interrupts and for now "down" is not being configured in this mode. To see the configuration options you can highlight the setting and press F12, which will take you to `stm32l4xx_hal_gpio.h` where the macros are declared.
- Initialize your selected pin.
- Test reading from your pin by using a tight polling loop in main to set the state of the green LED based on the state of the down pin.
- Copy your code into your project 1 report.

Success Condition:
You are able to determine the current state of joystick down.

A UVISION 5 CONFIGURATION

The Keil μVision5 IDE is the recommended environment for compiling and debugging the code for this project. Please follow these instructions to install the necessary packs for the project the first time you open it in uVision5. If the tools are not setup properly it's possible certain compile or run time errors will be encountered during the project development that can be hard to debug.

a.1 Installing the Necessary Packs

Before the IDE will recognize and add the required files for the micro-controller we are using, you need to download them to your IDE.

1. Open uVision 5
2. Click on the pack installer icon in the tool bar as shown in Figure 4.

Figure 4: Pack Installer Icon in Toolbar

3. You should now see something similar to the window shown in Figure 5. Click OK to close the information box.
4. In the Devices tab, search for "L476" and select "STM32L476VG". If this Device does not show up, go to Packs -> Check For Updates then try again after they load.
5. In the Packs tab on the right, click to install the Device Specific packages `Keil::STM32L4xx_DFP and Keil:STM32NUCLEO_BSP` .
6. If there are updates available for any of the generic packages already installed, install them as well. Note that you can tell it to install both packages from the previous step and all updates without having to wait for each one to finish.

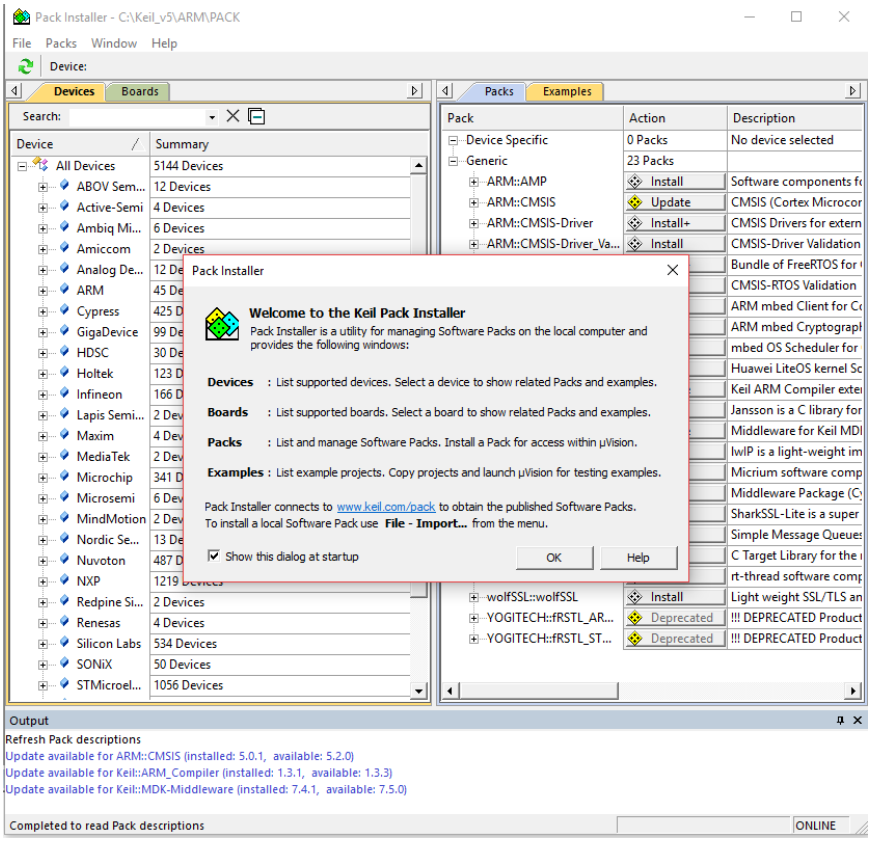

Figure 5: Pack Installer Welcome Screen

a.2 Creating a New Project

For this course, you should not need to do this. These instructions have been included just in case something corrupts or you want to create a new project for another purpose.

1. Project -> New uVision Project. Select a location for your project. It is recommended that there be no spaces in the file path and that you create your project in an empty folder.
2. Select the "STM32L476VGTx" device from the list and click OK.
3. For Board Support, select the STM32L476G-EVAL variant
4. You may wish to change "Target 1" to something more descriptive, like "L476VG" (optional)
5. You may wish to change "Source Group 1" to something more descriptive, like "User Files" (optional)
6. Copy the provided SRC and INC folders into your project folder (should contain a .uvprojx file)

7. You will either need to copy and add the provided driver files or select them from the "select software packs" window you can open from the toolbar.

8. Right click on “User Files” in the “Project” pane of uVision (left side) and select “Add Existing Files to Group” then select all the C files in the SRC folder you copied into the project and close the window.

9. Click on the “Options for Target” (looks like a wand) icon in the tool bar or Alt-F7, the select the “C/C++” tab. Copy the following into the “Include Paths” field: `.\INC`

B SYSTEM WORKBENCH INSTRUCTIONS

The following instructions are alternatives for the uVision specific instructions given in the manual. You will need to use these instead if you are using System Workbench. This IDE is only recommended if you are running MacOS or Linux.

b.1 Opening a Project

These instructions assume you have already installed the tools and downloaded the provided started code for git. If you have not done so, refer to Exercises 2.2 through 2.4.

Open System Workbench.You will be prompted to select a directory as a workspace. Navigate to "`Your_code_path\Project1\SW4STM32`" for Project 1. Replace the project folder as appropriate for later projects. If you are not prompted for a workspace and the tool loads an existing one, you can switch to the desired location by going to `File -> Switch Workspace`.

When it opens you should see a screen similar to Figure 6. If you aren't a fan of welcome screens, uncheck the box in the lower right corner so it doesn't show up next time. Close the welcome tab and you should see an empty project screen similar to Figure 7.

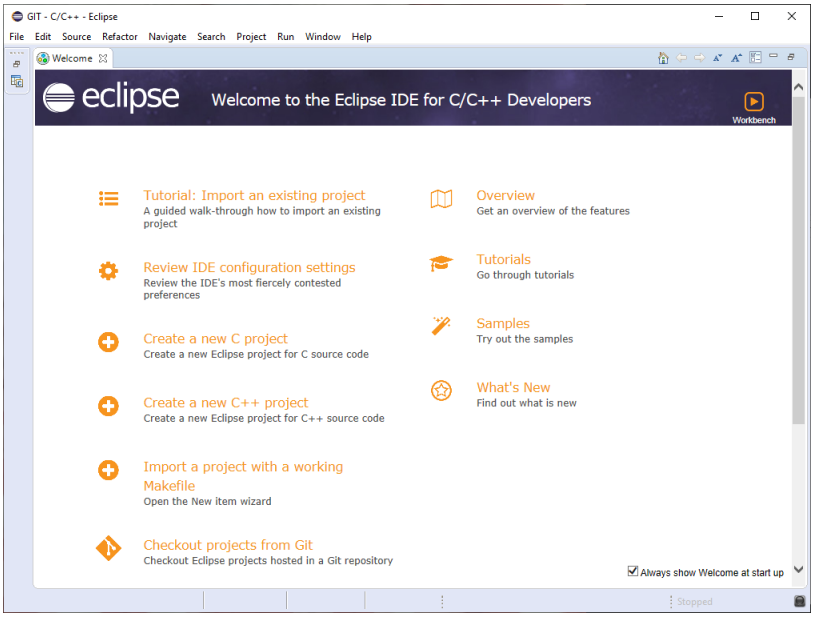

Figure 6: System Workbench Welcome Screen

To open the project, go to `File -> Open Projects from File System`. Use the `Directory...` button to browse to your workspace. There may be multiple folders listed, but only one will have "Eclipse project" in the "Import as" column. Check only this project, then click `Finish`. You should now have a project folder called STM32L476G-Discovery in the Project Explorer pane. Click the arrow be-

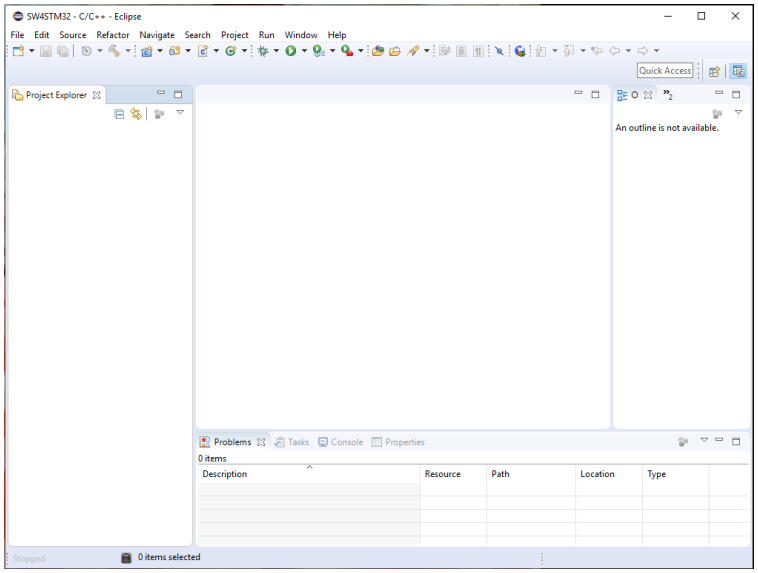

Figure 7: System Workbench Empty Project View

side it to expand it and see the source files for the project. Use Ctrl + B or click the Build All icon in the tool bar (white page with 010 on it). This may take a minute. Right click on the project folder in the Project Explorer pane. Select `Run As -> AC6 STM32 C/C++ Application`.

The provided starter code should now be loaded to the board. Verify that it is working by pushing straight down (select) on the blue joystick. The red led should turn on. Click the joystick again and it should turn off. If it is working, you can now begin working on your Project 1 code.

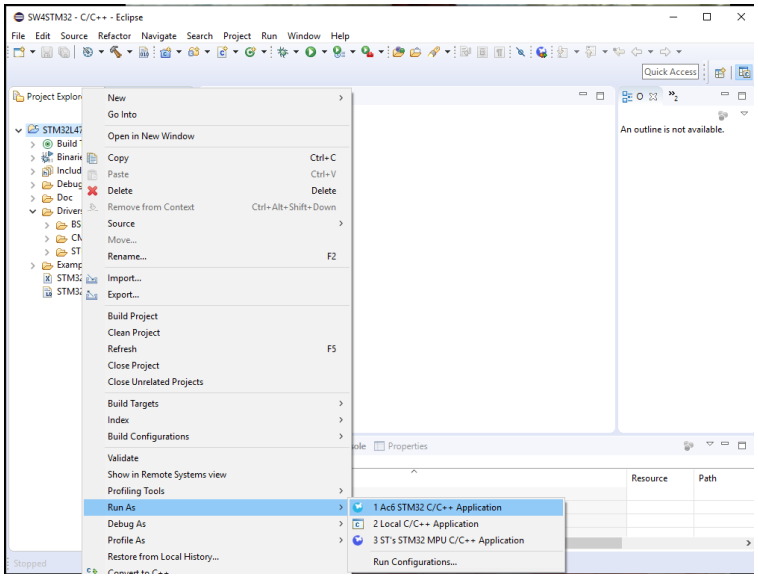

Figure 8: Selecting the Run Configuration

b.2 Debugging

You can start the debugger by clicking on the bug symbol in the main toolbar, going to `Run -> Debug` or using the `F11` key. You may be asked if you want to open the debug perspective. Click `Yes`. You should now see a view similar to Figure 9.

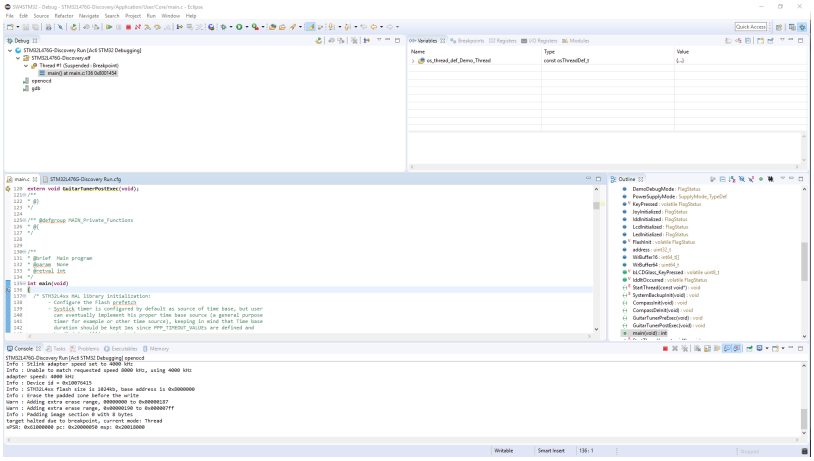

Figure 9: System Workbench Debug Perspective

Buttons 9 through 15 will allow you to work your way through the code. You can set breakpoints by clicking next to the line number. The `Variables` tab will show you the what is in scope and the current value when you hit a break point. To exit the Debug perspective, click the C/C++ view icon in the upper right corner.

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

Exercise B.1: Tools

Check Point 1 – Working Tools:

Before you write any new code for this project, you should confirm your tools are working by doing the following:

- Compile the provided code
- Load the code to the board
- Use the debugger to set a break point and check a variable (see the next section for tips on using the debugger)

Success Condition:
You are comfortable loading and debugging code on the board. Return to section 3.3 to continue.

C 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.

c.1 About Debugging

The Keil μVision IDE 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.

c.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 behavior 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.

c.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.

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

μVision Debugger, like many other modern debuggers, provides 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.

c.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.

c.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.

D DEBUGGER

d.1 Configuring the Debugger

The μVision debugger can be configured to `Simulator` or `Target debugger`. To choose one, right click on `STM32L476G-Discovery` in the `Project` window then select `Options for Target 'STM32L476G-Discovery' ...` or use `ALT+F7`. Click the `Debug` tab. The left side displays configuration options for the simulator while the right side displays the debugger options. When the board is connected, you should use the debugger. Click the `Use` radio button and pick `ST-Link Debugger` from the corresponding drop down menu at the top right. Do not change the other settings.

d.2 Using the Debugger

After successfully compiling your code, click the `Start/Stop Debug Session` button or press `CTRL + F5` to start the debugger.

Figure 10: The icon that looks like a magnifying glass with a d in it is the debug start/stop button

When the debugger starts, additional windows will open in μVision. You should see something similar to Figure 11 depending on the view configuration.

The debug controls are shown in Figure 12. The first 7 buttons from the left 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 in uVison will show a tool tip explaining the functionality of the button.

Buttons 9 though 19 control what windows are visible in your debug view. The first time you start the debugger, you will likely have additional windows visible. Holding your mouse over each icon will open a tool tip with more information. You may want to enable/disable the windows to match what is shown in Figure 12. This is simply a recommended starting point that highlights the important aspects of the debugger to help you with this course. Time permitting, you may want to play with the other views and see what information they can provide. Ultimately, the views used and the arrangement of your debug view will be a matter of personal preference.

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

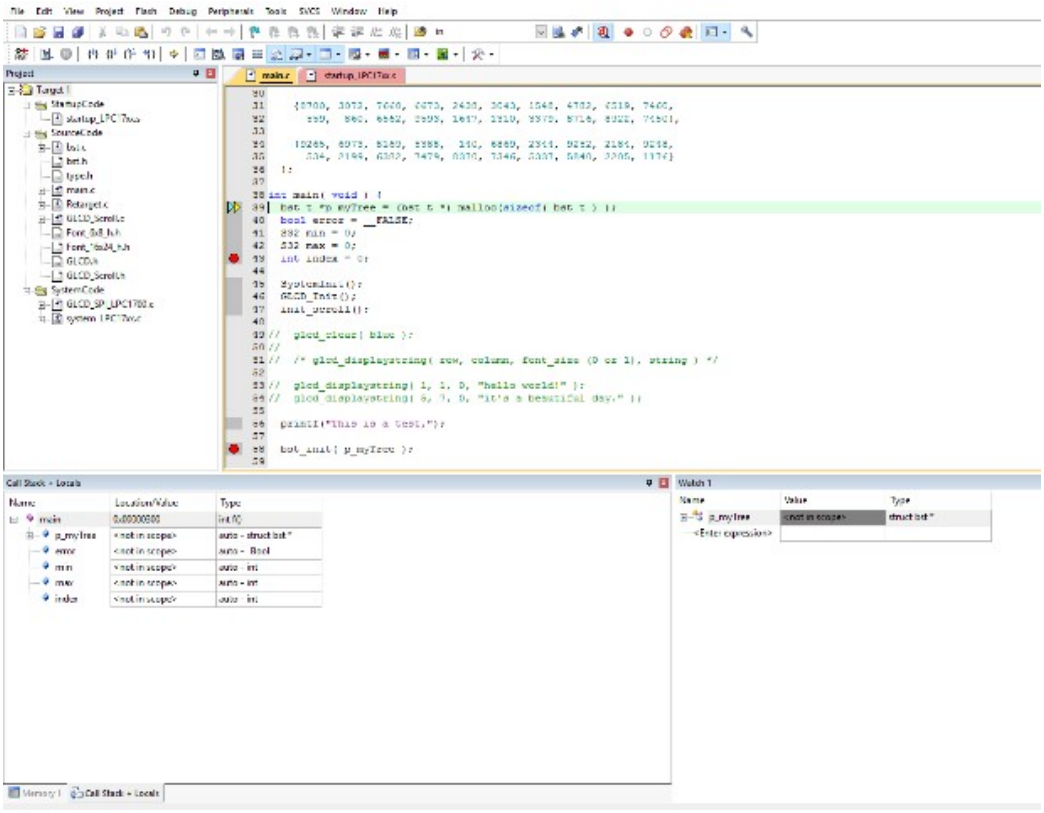

Figure 11: The debug view. Yours may appear slightly different depending on which windows are enabled and how they are arranged.

Figure 12: Debug Control Toolbar

d.3 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.

μVision supports execution, read/write access, and complex breakpoints. An execution 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 go to Debug menu and select Insert/Remove Breakpoint. You can also 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 Debug menu 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.

d.3.1 *Call Stack and Local Window*

The Call Stack and Locals window 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 13.

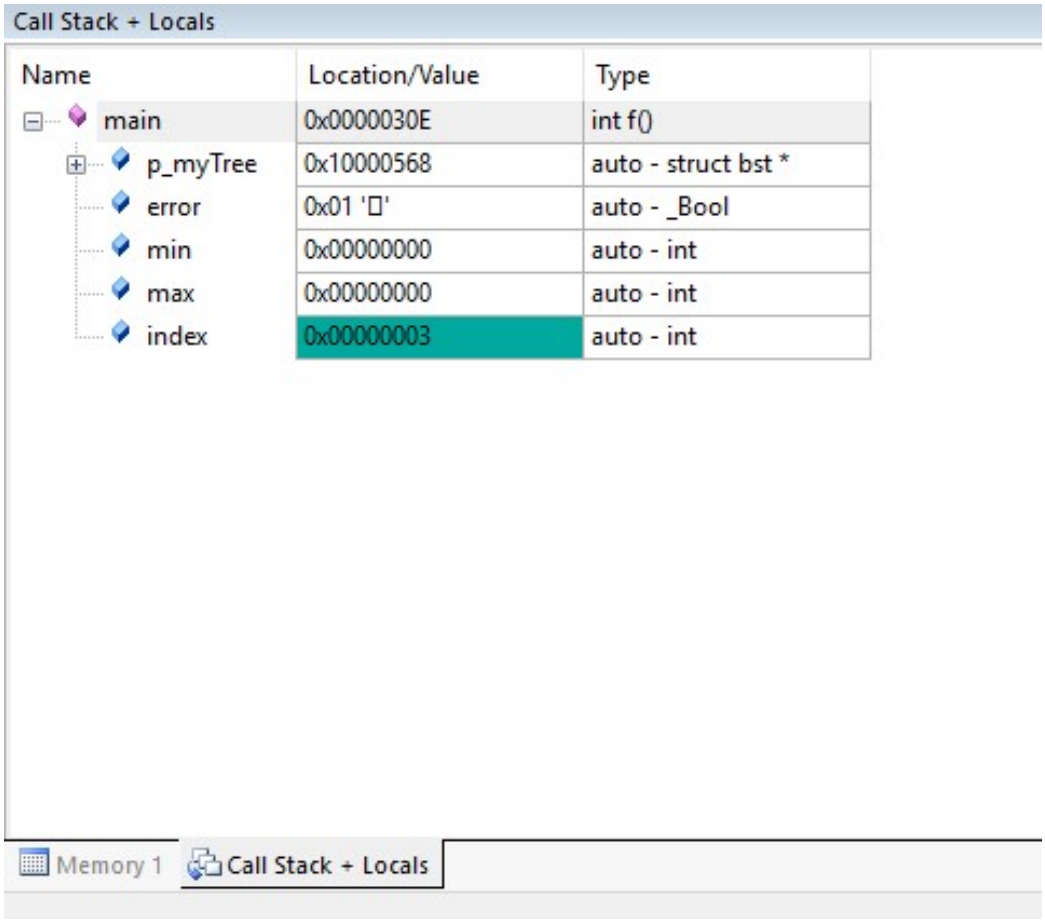

Figure 13: Sample view of the call stack and local window

d.3.2 *Watch Windows*

Using the watch window, one can see the content of any variable at any time while the execution is stopped. To watch a variable, select the variable, right click on the selected name, and choose `Add to watch1`. The content of the variable becomes visible whenever the variable is in scope within the current trace. An example is shown in Figure 14. The `p_myTree` node has been expanded to show how this view

can be used to trace the location and contents of each node in the tree. Notice that `root` has both a location, where it is physically located in memory, and a value.

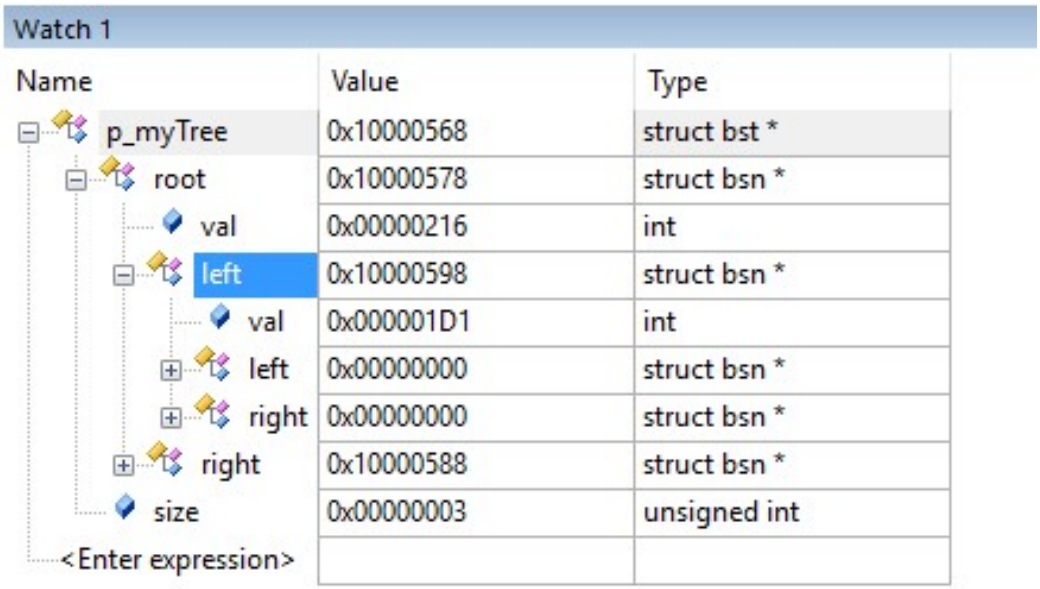

Figure 14: Sample watch window for binary tree program

d.3.3 *Memory Windows*

The memory window allows you to inspect the current contents of memory while program execution is stopped. Enter an address in the `Address` box at the top of the window. The memory contents starting from that address are displayed. Right click on the data to control the display format.

Given an address location, the memory window shows the data as bytes or words depending on the data format selected. The memory content can also be shown in hexadecimal or decimal. As shown in the following figure, by right clicking in memory window opens a menu from which many formats can be selected. The appropriate choice will depend on the type of data you are trying to review. If your data is a string array, choose `Unsigned->Char`. If your data is an array of unsigned integers, choose `Unsigned->Int`.

Acknowledgments

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 it has been further updated for uVision 5 and MTE 325.

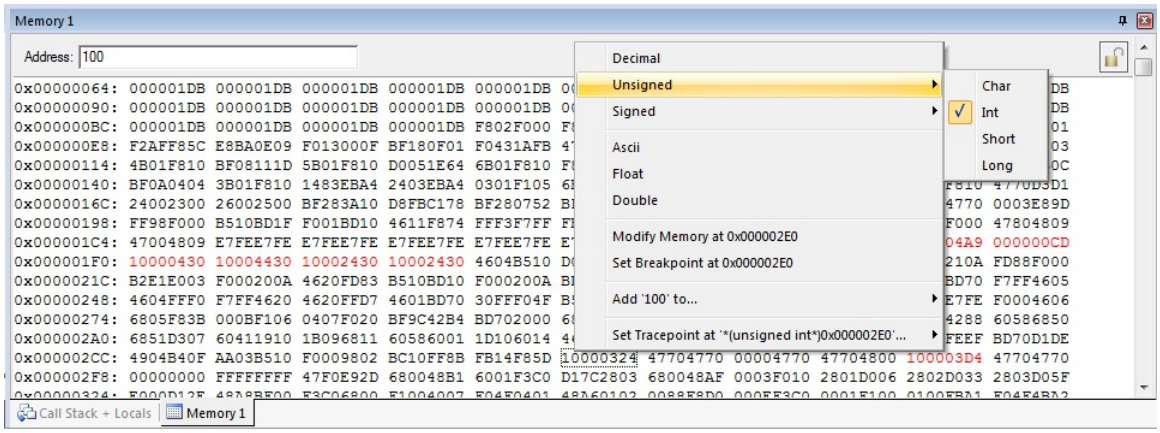

Figure 15: Sample Memory Watch window with settings menu open

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.