

MTE 241 – LAB 1

Created by: Allyson Giannikouris, P.Eng.

Last updated by: Bernie Roehl, Fall 2018

1 Introduction

This lab has been designed to introduce the tools you will be working with for your MTE 241 labs. The Introduction and Background sections should be reviewed before the start of your lab session. You are expected to complete all exercises within your 3 hour lab session. The information contained in the Appendices is applicable to all labs in this course.

Learning Objectives

After completing all 4 exercises, you should be able to:

- Identify and fix C syntax errors
- Understand and fix common compile time errors
- Identify code that does not conform to common C formatting standards and make the appropriate changes
- Print to a terminal window
- Use the debugger to
 - Watch local and global variables
 - Check the contents of memory
 - Set breakpoints
 - Step through code

2 Background

What you need to know before you start

The tools we are using for this course can be divided into two categories: hardware and software.

The hardware is the physical Keil MCB 1700 demo board which should be attached to your lab computer. The board is shown in

Figure 1. Take a moment to look at it. What components do you recognize? Which parts are you unsure of the purpose of? For the first three labs in this course, we are only really concerned with two components on this board: the microprocessor and the reset button. The microprocessor will run your code, while the reset button has to be pressed to start your program when not using the debugger. You will get a chance to learn more about and familiarize yourself with other components on the board in subsequent labs.

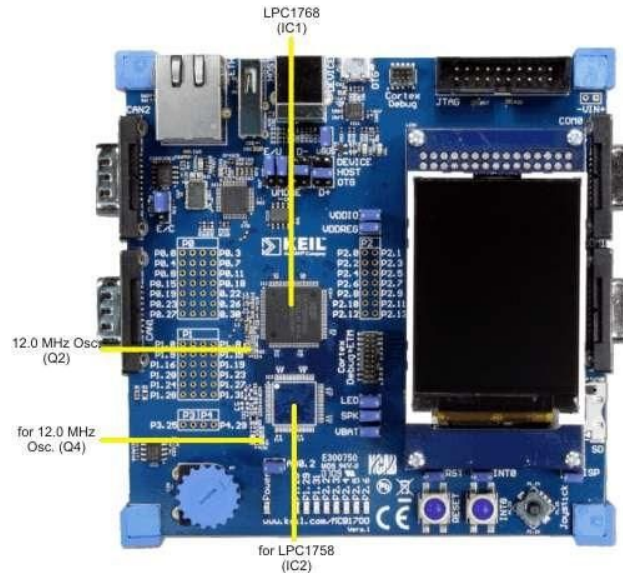


Figure 1: The Keil MCB 1700 evaluation board with microcontroller location indicated

The second key tool you will use in this course is the Keil μ Vision 5 integrated development environment or IDE software. This is the tool you will use to compile your code into a form that can be loaded and run on the microcontroller. It is also the main tool that you will use to debug your code.

2.1 About Debugging

The Keil μ Vision 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 localizing, understanding, and correcting 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 understanding of 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 the debugging procedure. This step also includes verification to ensure the fault is mitigated and no new issues have been introduced.

2.2 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.

2.3 Functional Debugging

Given a test-case revealing a fault in the 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 localizing and understanding become 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.

2.3.1 *Tracing the Code Step-by-Step*

The μ Vision 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 when 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.

2.3.2 *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 a very powerful debugging tool that are used to stop the execution on a 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 which is suspected as the root cause of a fault.

2.3.3 *Instrumenting with printf*

Using `printf` statements is probably the most 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 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.

2.4 About the Source Code

Before proceeding any further, you should go through the steps described in Appendix A.

The Lab 1 source code that has been provided to you implements a binary search tree. You may wish to search online or consult an algorithms textbook to review the fundamentals of this data structure.

Table 1 provides a list of the files contained in Project1_source.zip. Those in bold are the files specific to Lab 1.

Table 1: List of files contained in P1_Source.zip

| File Name | Description |
|------------------|--|
| bst.c | The binary tree data functionality (class) is implemented in this file. |
| bst.h | The associated header file for bst.c. Contains the function prototypes and defines the bsn_t and bst_t structs. |
| p1_main.c | The main function is implemented here. This file also contains code to test the binary tree data structure. The code performs a series of insertions followed by a series of deletions to verify the implementation. |
| Retarget.c | This file is needed to direct the printf outputs to the terminal. The details of this file are not important for Lab 1. |
| type.h | Contains definitions for the types used in this, as well as all future, labs. |
| uart.c | Implements a Universal Asynchronous Receiver Transmitter (UART). The details of how this is done are not important for Lab 1. |
| uart.h | The associated header file for uart.c containing function prototypes and definitions. |

The binary search tree struct stores the address of the root node and the number of nodes currently stored in the tree. This is implemented using the `bst_t` structure, as defined in `bst.h`.

Every node within the tree stores the addresses of both the left and right children (NULL if there is no child there) as well as the integer value stored in this node (of type `S32`). This is implemented using the `bst_n` structure, as defined in `bst.h`.

The `bst.c` file implements the following functions (with some errors you will need to fix) to manipulate the binary search tree with n nodes and a height h :

`void bst_init(bst_t *);`

Initialize the binary search tree so that it is empty. Run time: $\Theta(1)$

`U32 bst_size();`

Return the number of nodes in the binary search tree. Run time: $\Theta(1)$

`bool bst_insert(bst_t *, S32);`

Insert the given integer into the binary search tree and return false if the node is already in the tree (do not add a duplicate into the tree) and true otherwise. Run time: $O(h)$

`S32 bst_min(bst_t *);`

Returns the smallest integer in the binary search tree. Return `INT_MAX` if the tree is empty. Run time: $O(h)$

`S32 bst_max(bst_t *);`

Returns the largest integer in the binary search tree. Return `INT_MIN` if the tree is empty. Run time: $O(h)$

`bool bst_erase(bst_t *, S32);`

If the object is in the binary search tree, remove it and return true; otherwise, return false and do nothing. Run time: $O(h)$

While completing the various exercises, you are welcome to create other helper functions and you are welcome to add additional fields onto any of the records as you find necessary.

`#include <stdbool.h>` has been added to allow access to the type `bool`.

`#include <limits.h>` is used to access `INT_MIN` and `INT_MAX`.

3 C syntax

Simple typos and beyond

The compiler's job is to check that your code conforms to the rules of the C programming language, and if it does, convert it to machine code. If your code compiles successfully, a file with a `.elf` extension will be produced. The tool chain will then convert this `.elf` file to a machine code file with the extension `.axf`.

When there is a problem, the best case scenario for the programmer is a meaningful error message or warning. These are the easy bugs to fix. Unfortunately, there will be times where the error message is seemingly unrelated to what is found to be the root cause of the problem. While you will likely come across such errors at some point, today is not the day.

Exercise 1: Fix the syntax errors.

Click the compile button. If your project was properly configured and all necessary files properly added, you should have 30 errors. Read the errors and correct the 5 problems that are causing them.

Success Condition:

The following should be the last line printed to the "Build Output" window:

"Project1.axf" - 0 Error(s), 0 Warning(s).

4 C formatting and style

Nobody likes a monster function

Making code that compiles often isn't enough. Making code that is readable, maintainable and can easily be verified and updated is important too.

You should have successfully compiled the provided code, but...

1. Is it easy to read?
2. Is it easy to test?
3. Can you read it and understand what each function/line/loop is doing?

Exercise 2: Formatting and Style

Make 5 improvements to your code to improve the formatting and style.

So your code compiles, but it's ugly. Clean it up.

You may use the formatting and style guidelines discussed in class or any other generally accepted C guidelines. While commenting is an important part of good coding style, your changes should not be the addition of comments for this exercise. There are many possible improvements that can be made to the sample code, you need to make 5, but may choose to make others time permitting.

Success Condition:

Make the changes to your source code. Be sure to document your changes in the summary block at the top of "bst.c".

5 Terminal Communication

Hello World and beyond

One tool often used for debugging and program verification, or for runtime input and output from the user, is a terminal window on the host PC. The terminal is a separate program that runs on the PC that the microcontroller communicates with. In our case, this is done through a serial (i.e. COM port) connection.

5.0.1 *What your project needs*

The source code you were provided with is able to communicate with the terminal program because the following files are included in the project:

1. uart.h
2. uart.c
3. retarget.c

You need to tell the program you want the input/output to be directed to the terminal window and not the LCD. This is done using a C pre-processor symbol.

Open the Options for Target ... window (ALT+F7) and select the C/C++ tab. In the panel for Preprocessor Symbols, in the box marked Define, add `__RTGT_UART` as shown in Figure 2. Note the double underscore at the start of the identifier.

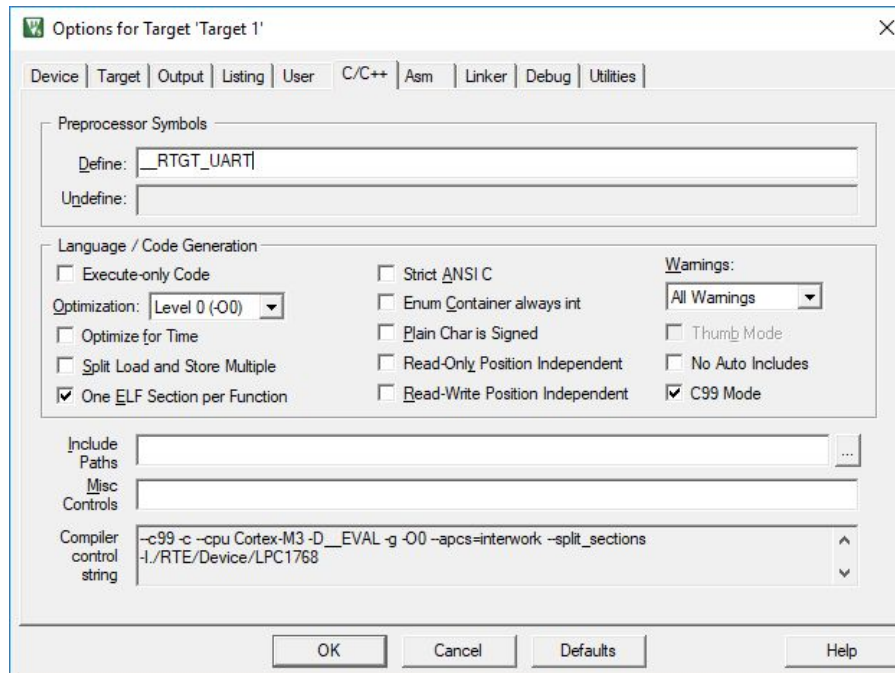


Figure 2: The C/C++ tab of the Options for Target window with the `__RTGT_UART` pre-processor #define added

5.0.2 Launching PuTTY

There are many terminal programs available. The instructions given here are for PuTTY, but you are free to use others if you prefer.

The easiest way to find PuTTY is to search for it in the Windows Start menu in the "Search Programs and Files" text box.



Figure 3: PuTTY Icon

When you launch PuTTY, the PuTTY Configuration dialog opens as shown in Figure 4. Under Session, select a Connection type of Serial and set the speed to the baud rate you have defined in your source file (for Lab 1, 9600). The changes you need to make are highlighted.

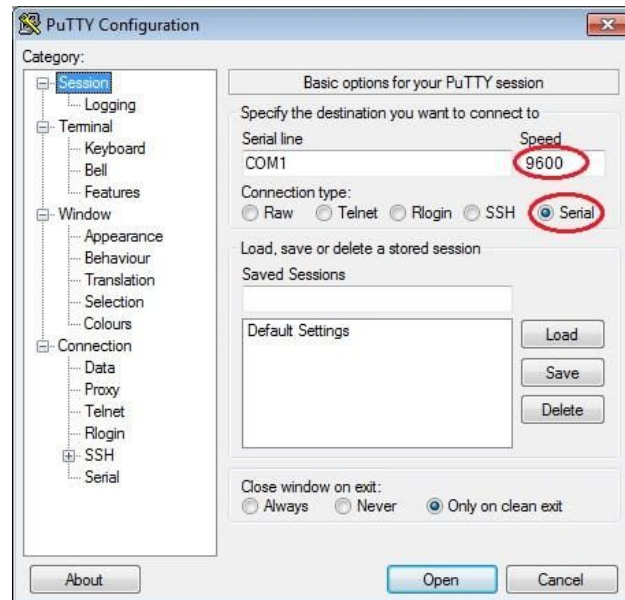


Figure 4: The PuTTY Configuration Window. Select the settings shown.

You will also need to set the COM port number. The Windows COM port that's connected to the Keil board varies from station to station, and may also change from one session to another on a particular station. In order to find out the COM port to use, open a CMD window and type the following command:

```
Q:\eng\ece\util\find-com
```

The output from that program will include something like COM5 or COM7. That's the port you specify in Putty.

If for some reason that doesn't work, you can use the Windows Device Manager. Hold down the Shift key and right-click on the Windows icon at the lower left of your screen. Select Device Manager and click OK to the warning. Expand the "Ports (COM &

LPT)” entry and look for the COM port. That’s the port you specify in Putty.

There is one more setting that must be changed for the Lab 1 configuration. Click on Serial in the left menu bar, and set Flow control to None as shown in Figure 5.

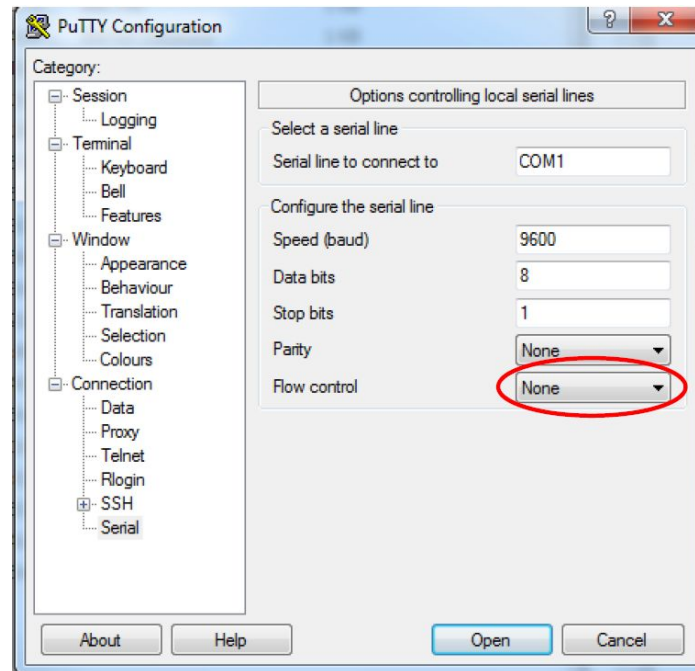


Figure 5: Flow control configuration in PuTTY

Click Open. This will open a window similar to the one shown in Figure 6, although you won’t see any text printed yet. This is where text printed using `printf` in your code will be displayed. It is also where user input can be given to the program when requested by the code, although we won’t use this feature in Lab1.

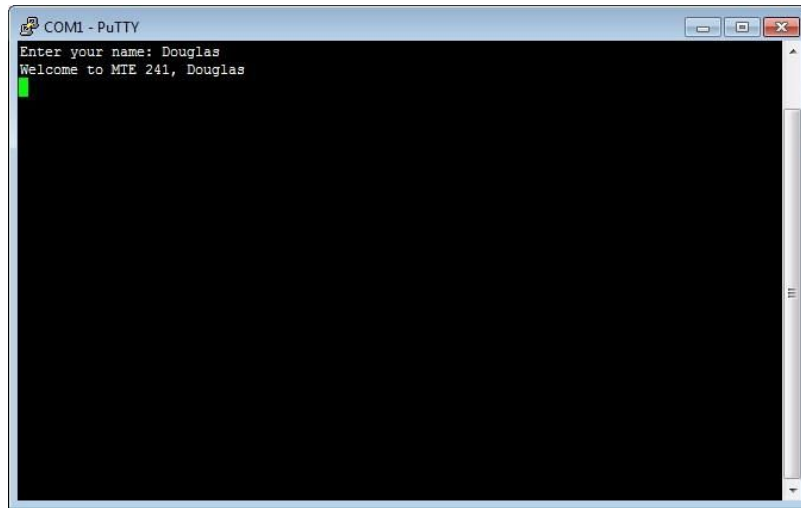


Figure 6: Sample PuTTY terminal window

You should now be able to compile your code and load it in to the board. After hitting the reset button on the board with the terminal running, you should see text printed to the screen. See [Section A.4](#) for instructions on how to load your code to the board and start your program.

Exercise 3: Hello Binary Tree

Verify that you are able to print to the terminal window.

Follow the steps above to add the required preprocessor directive to your project settings. Open a terminal window, recompile and run your code.

Success Condition:

You should see text in the terminal window (and it's not just a bunch of random symbols).

6 Logic bugs

It compiles, but does it work?

The Background section provided a description of debugging, as well as some commonly used debugging techniques. Appendix [B](#) provides an overview of the most relevant features of the Keil debugger for this project. Try to familiarize yourself with all of them while completing this exercise.

Exercise 4: Fixing Logic Bugs

Verify that the code implements a binary tree correctly.

Code that runs is the first step, now you need to make sure it runs properly. Use the debugger to help you verify that the provided code gives the output shown in Table 2. If you find it does not, use the debugger to help you fix the logic. Document your bug fixes in the corresponding header section, including a note on the debugger features you used to help you.

Success Condition:

The output shown in the terminal window when you run your code has the values shown in Table 2 after each round of deletions.

| | Min | Max |
|----------------------------|------------|-------------|
| Before first group erased: | -3 | 9593 |
| After first group erased: | -3 | 9593 |
| After second group erased: | 23 | 9593 |
| After third group erased: | 140 | 9593 |
| After fourth group erased: | 140 | 9265 |
| After fifth group erased: | 2147483647 | -2147483648 |

Table 2: Expected min and max values after each deletion cycle

7 Marking

To receive credit you will need to do the following:

1. Upload your `bst.c` and `p1_main.c` files to LEARN. Don't forget to fill in the header sections summarizing your formatting and logic fixes in `bst.c`.
2. Demonstrate your knowledge to a TA before the end of the lab session. You will be asked to show two of the skills described in the objectives section. You can use this manual as a reference during the demonstration.

8 Going Further

The following resources may be of interest to those of you looking to reinforce or extend the skills you have learned in this lab.

- Doug Harder's Website contains material from a previous offering of the course. He has pages on several aspects of the Keil environment you may find useful while completing labs in this course.

https://ece.uwaterloo.ca/~dwharder/icsrts/Keil_board/

- The Keil website contains more information about both the board itself and the uVision environment.

<http://www.keil.com/support/man/docs/mcb1700/>

A Project Configuration

The Keil μ Vision 5 IDE is used to compile the code for all projects in this course. It is important to set up the project and the environment properly before you start coding each project. 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 Setting Up the Environment

On your N\ drive create the path N:\MTE241\Lab1

Start the Keil μ Vision 5 application and select Project->New uVision Project....

Navigate to N:\MTE241\Lab1 and save the project as 'Lab1' as shown in Figure 7. **It's a bad idea to use spaces, periods or other unusual characters in your path.** It's also a bad idea to create your project folder on the local computer. It may not be there when you come back for it and/or you provide an opportunity for others to use your code without your knowledge. Use your N: drive instead.

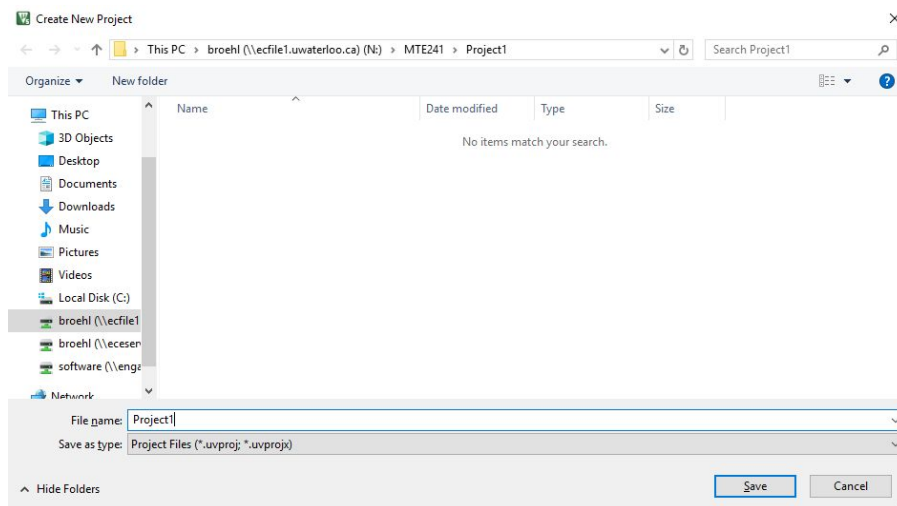


Figure 7: Select a location to save your project

The tool needs to know which device you are compiling for.

This course uses the Keil MCB1700 demo board, which has the **NXP LPC1768** microcontroller on it. Ensure this device is selected, as shown in Figure 8 then press OK.

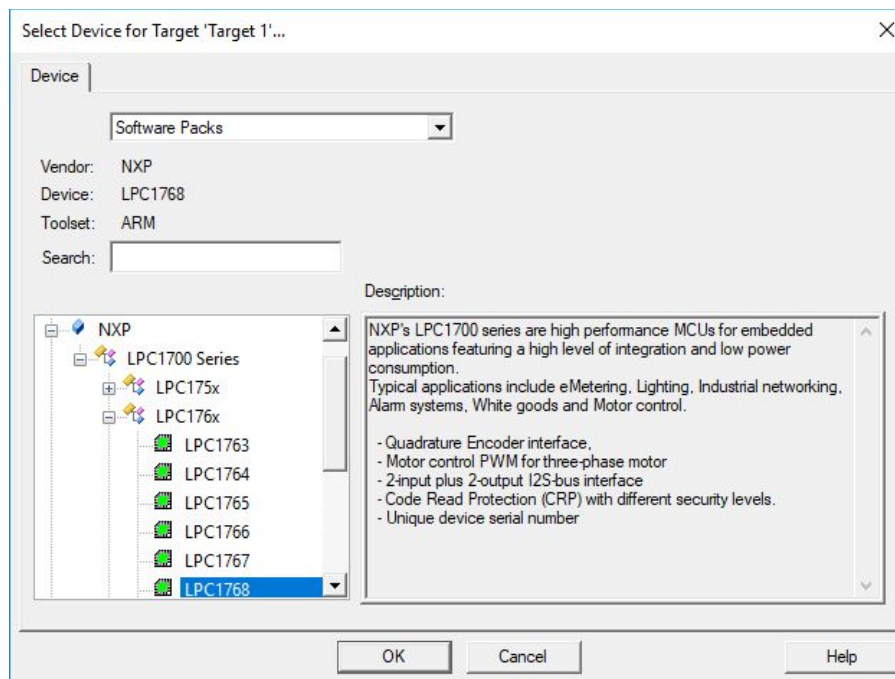


Figure 8: Select the NXP LPC 1768 as the device for this course

In the dialog box that appears, expand the Device element and click on the checkbox next to “Startup” as shown in Figure 9. Expand the CMSIS element and click on the checkbox next to “CORE”. Click on OK to finish creating the project.

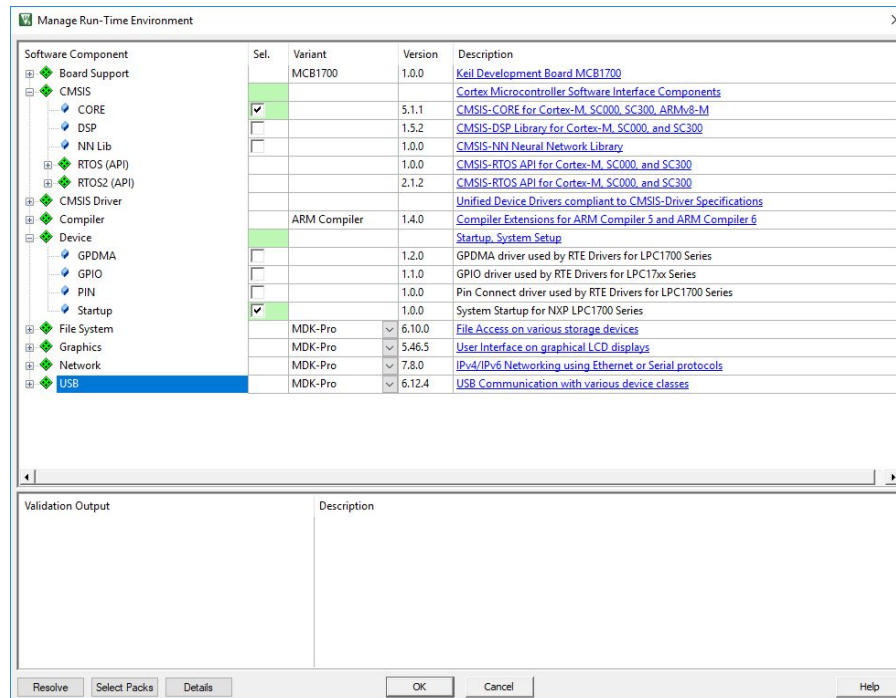


Figure 9:Adding components

The startup file contains a number of details that describe the device we are using and its capabilities to the tool chain. You can find it in your Project window on the left side of the screen by clicking on the + next to Target 1 then on the + next to Device. One important setting we need to change is the stack size. This will help to ensure your code doesn't run out of memory while executing the project. Double click on startup_LPC17xx.s and click the Configuration Wizard tab as shown in Figure 10.

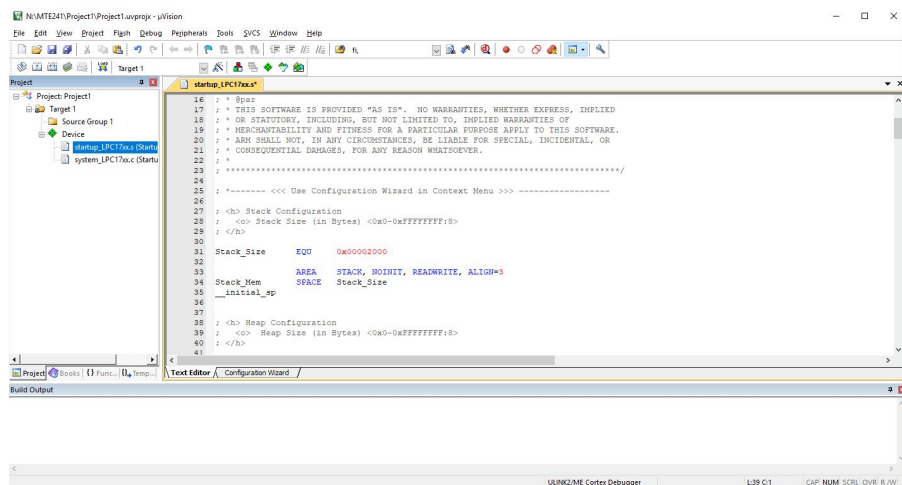


Figure 10: Accessing the startup configuration settings

Change the Stack Size to 0x2000 as shown in Figure 11, which is certainly more than what we need (the default is not enough for Lab 1). Click the plus sign next to Heap Configuration. Change the Heap Size to 0x2000 as well to allocate the dynamic memory you will need. Click the save icon on the top menu bar.

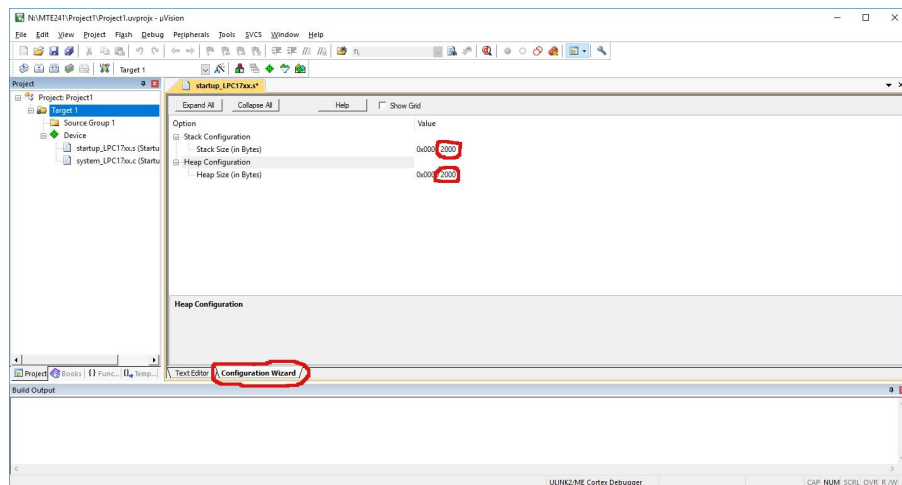


Figure 11: Changing the stack configuration

We want to enable use of the built-in malloc function to allocate memory. To do this, the target options need to be set appropriately. Right click on Target 1 and select Options for Target 'Target 1'... or use ALT+F7.

The window shown in Figure 12 should now be open. Ensure the Use MicroLib checkbox is checked.

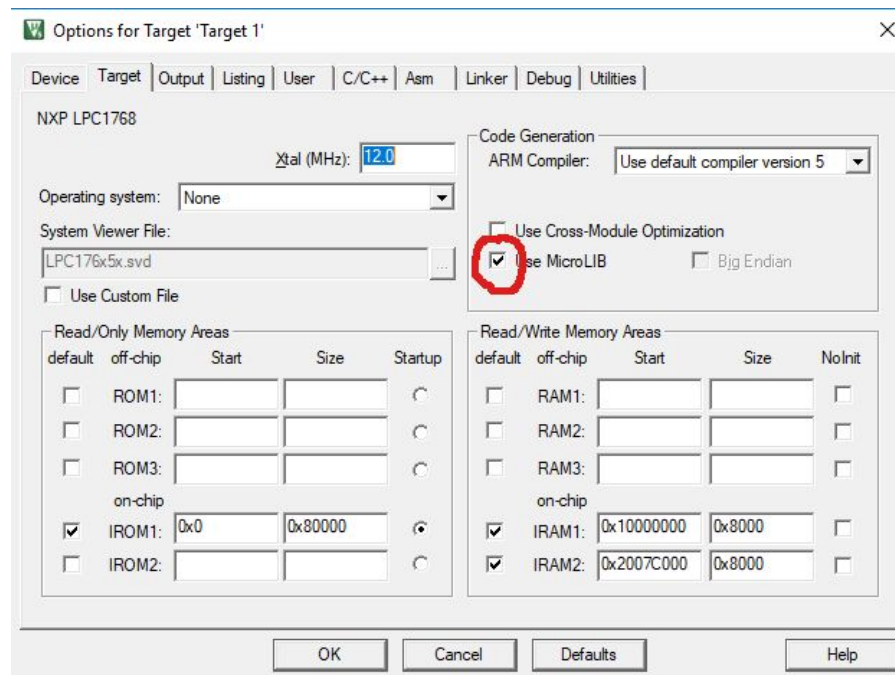


Figure 12: Enabling the MicroLIB library

Take a quick look at the Debug tab, and make sure that in the right-hand column the “Use” radio button next to “ULINK2/ME Cortex Debugger” is enabled, as shown in Figure 13. When you’re done, click the OK button.

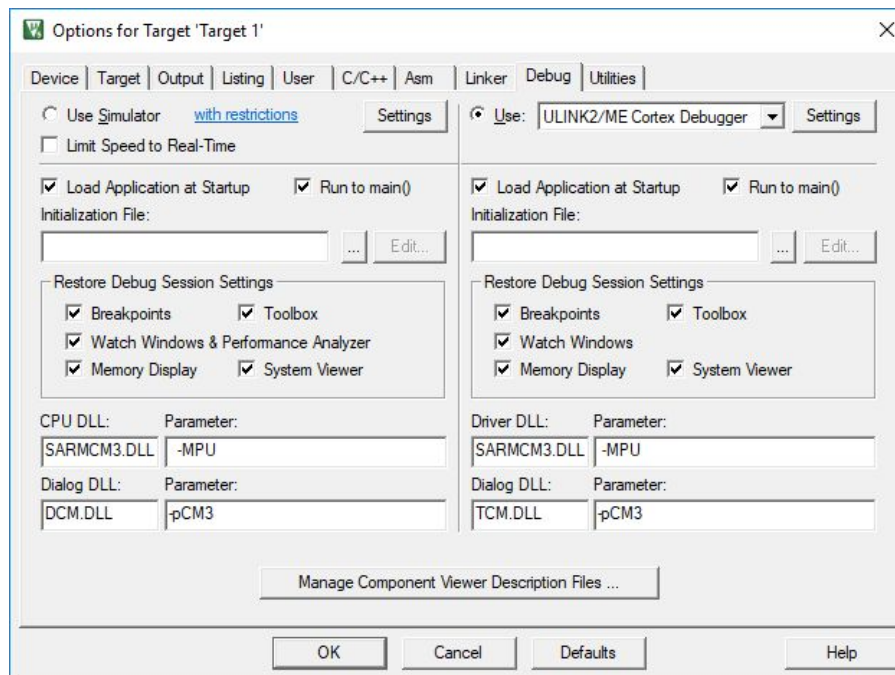


Figure 13: Make sure you're using the Cortex debugger

A.2 Adding files to your project

Now we can add the project files we need. For Lab 1 you will need to download the MTE241_P1_source_code.zip file from LEARN and unzip it into your project directory (N:\MTE241\Lab1).

Now you need to add the appropriate source files to your project. Right click on Source Group 1 and use the "Add existing files" item to add the four source files and then click "Close". Your Project window should now look like Figure 14.

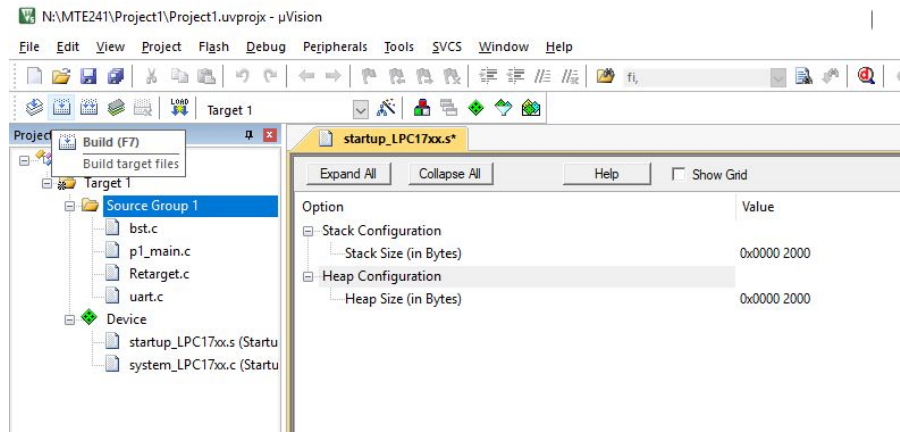


Figure 14: What your project should look like after adding source files

A.3 Creating the Main File and Compiling

For Lab 1, a main file has been provided called `p1_main.c`. This file contains a function called `main`, which the compiler will interpret as the entry point into your program.

At this point you should be able to compile your project by clicking the Rebuild All button. Note that there are two rebuild buttons, as shown in Figure 15.

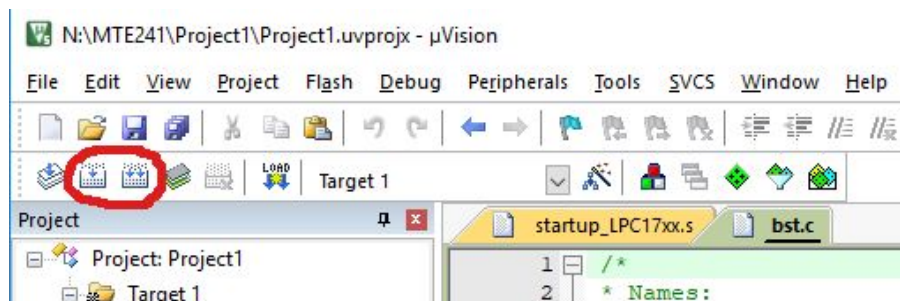


Figure 15: Location of the rebuild buttons on the toolbar

The one on the right, with two arrows, rebuilds everything. The one on the left, with a single arrow, rebuilds (i.e. compiles) only the current file. This can save you some time if you're only making changes to one file.

For Lab 1, compiling should fail. It is your job to correct the errors. Now is a good time to go back to Section 4, C Formatting and Style. Once your code compiles, you can continue to the next section in order to download and run your code. Note that you can double-click on errors in the console to jump to the lines in your source code. Note that you will need to set the `__RTGT_UART` variable as described there, regardless of which lab you're doing in this course.

A.4 Downloading and Running Code on the Board

When your code is compiling correctly, the build output window at the bottom should say something like

"Project1.axf"- 0 Error(s), 0 Warning(s)

If it says something else, you will need to resolve the issues before proceeding.

Download the code to the board by pressing the Load button as shown in Figure 16.

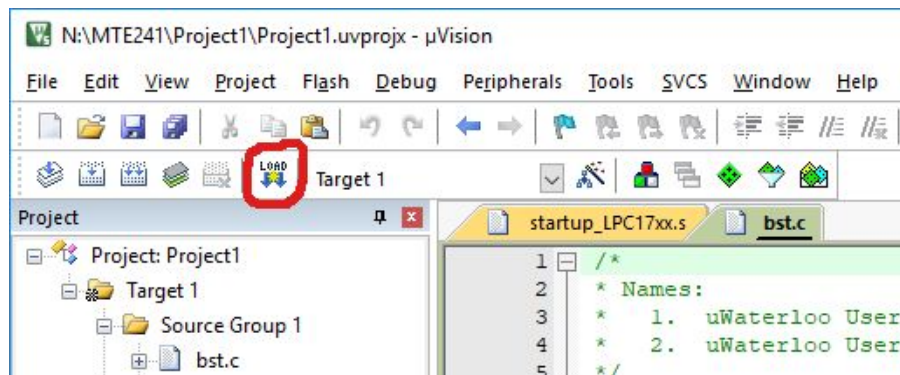


Figure 16: Location of the load button on the toolbar

To start the processor, press the reset button on the Keil MCB1700 board as shown in Figure 17.



Figure 17: Location of the reset button on the Keil board

If everything worked correctly and you have a terminal window connected, you should see something similar to Figure 18.

Acknowledgements

The material in this section was adapted from a document titled "*Project 1 - Tool Chain Tutorial*" that was prepared for the Fall

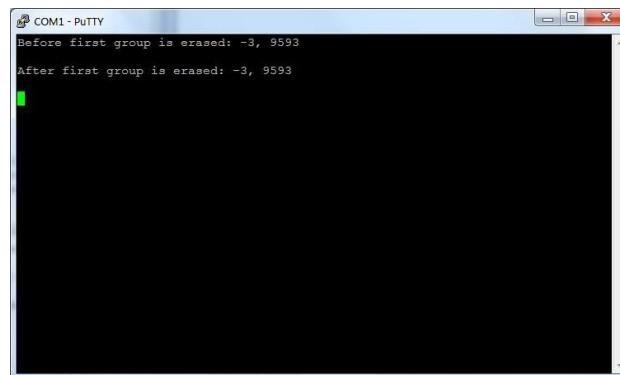


Figure 18: Output displayed in Putty

2015 MTE 241 course offering by Trevor Smouter with acknowledgments to Doug Harder and Allyson Giannikouris.

B Debugger

B.1 Configuring the Debugger

The μ Vision debugger can be configured to Simulator or Target debugger. To choose one, right click on Target 1 in the Project window then select Options for Target 'Target 1' ... or use ALT+F7. Click the Debug tab. You should now see a window similar to what is shown in Figure 19. The left side displays configuration options for the simulator while the right side displays the debugger options. For Lab 1, we will be using the debugger. Click the **Use** radio button and pick ULINK2/ME Cortex Debugger from the corresponding drop down menu at the top right. Do not change the other settings.

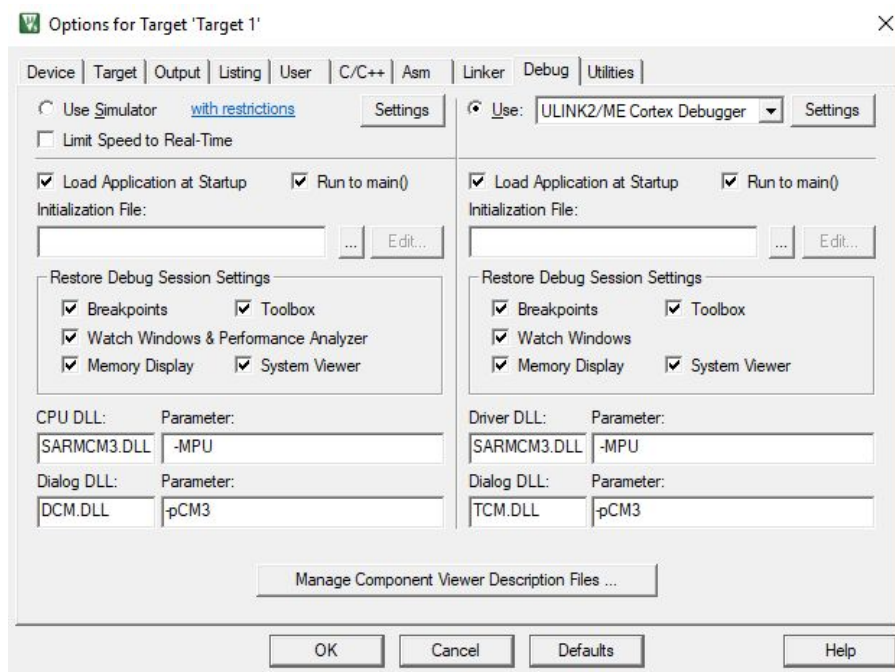


Figure 19: Debugger Configuration Menu

B.2 Using the Debugger

After successfully compiling your code, click the Start/Stop Debug Session button as shown in Figure 20 or press CTRL + F5 to start the debugger.



Figure 20: 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 μ Vision. You should see something similar to Figure 21 depending on the view configuration.

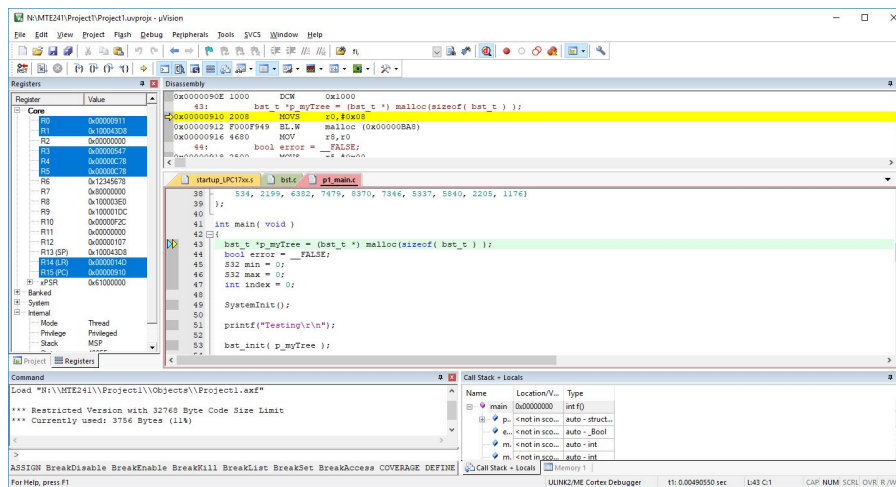


Figure 21: The debug view. Yours may appear slightly different depending on which windows are enabled and how they are arranged.

The debug controls are shown in Figure 22. The first 8 buttons from the left allow you to start the program, stop the program and step through the code in a few different ways. Holding your mouse over each of the icons in uVision will show a tooltip explaining the functionality of the button.



Figure 22: Debug Control Toolbar

Buttons 9 through 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 tooltip with more information. For Project 1, you

may want to enable/disable the windows to match what is shown in Figure 22.

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 Lab 1, we will introduce hardware breakpoints, and the Call Stack Window, Watch Windows and Memory Windows.

B.3 Hardware Breakpoints

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 breakpoints.

μ Vision 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 the 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 there when the program is run. Breakpoints can be disabled or removed via the 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, register 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. You can also use the Reset button to restart the execution.

b.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 23.

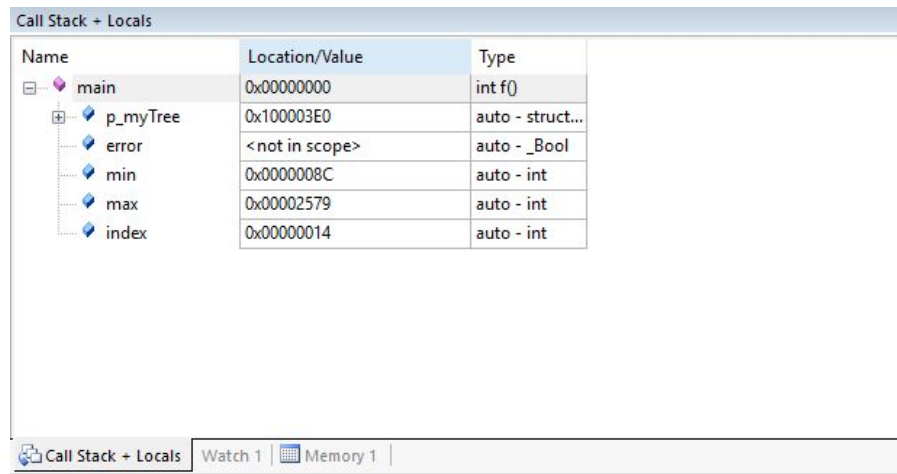


Figure 23: Sample view of the call stack and local window

b.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 showing the binary tree from Project 1 is shown in Figure 24.

| Watch 1 | | |
|--------------------|------------|--------------|
| Name | Value | Type |
| p_myTree | 0x100003E0 | struct bst * |
| root | 0x100003F0 | struct bsn * |
| val | 0x00000216 | int |
| left | 0x10000A20 | struct bsn * |
| val | 0x0000008C | int |
| left | 0x00000000 | struct bsn * |
| right | 0x00000000 | struct bsn * |
| right | 0x10000400 | struct bsn * |
| val | 0x000018EE | int |
| left | 0x10000420 | struct bsn * |
| right | 0x10000430 | struct bsn * |
| size | 0x00000014 | unsigned int |
| <Enter expression> | | |

Call Stack + Locals
Watch 1
Memory 1

Figure 24: Sample watch window for the binary tree program

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 each variable has both a location, where it is physically located in memory, and a value.

b.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. (Hint: for Lab 1 the start address of the memory you are managing is a good place to start). 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.

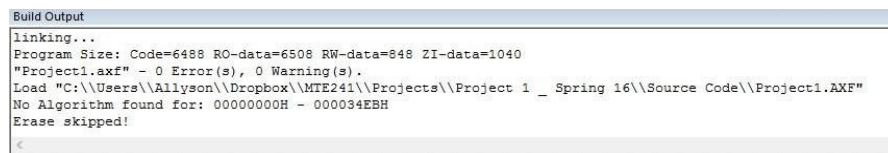
For later projects, you may choose to make use of the simulator to test your code outside of the lab.

C Common errors and how to fix them

Some of the more commonly encountered errors are described here. Often these are the result of copying the project from one location to another.

c.1 No Algorithm Found

You will get this error when you try to load your code to the board, with an error message in the Build Output window as shown in Figure 26. The algorithm it is referring to is the programming algorithm.



```
Build Output
linking...
Program Size: Code=6488 RO-data=6508 RW-data=848 ZI-data=1040
"Project1.axf" - 0 Error(s), 0 Warning(s).
Load "C:\\Users\\Allyson\\Dropbox\\MTE241\\Projects\\Project 1 _ Spring 16\\Source Code\\Project1.AXF"
No Algorithm found for: 00000000H - 000034EBH
Erase skipped!
```

Figure 26: Sample "No Algorithm Found" error message

Solution:

Open the "Options for Target.." menu (ALT+F7) and select the Utilities tab. You should see what is shown in Figure 27. Click on Settings.

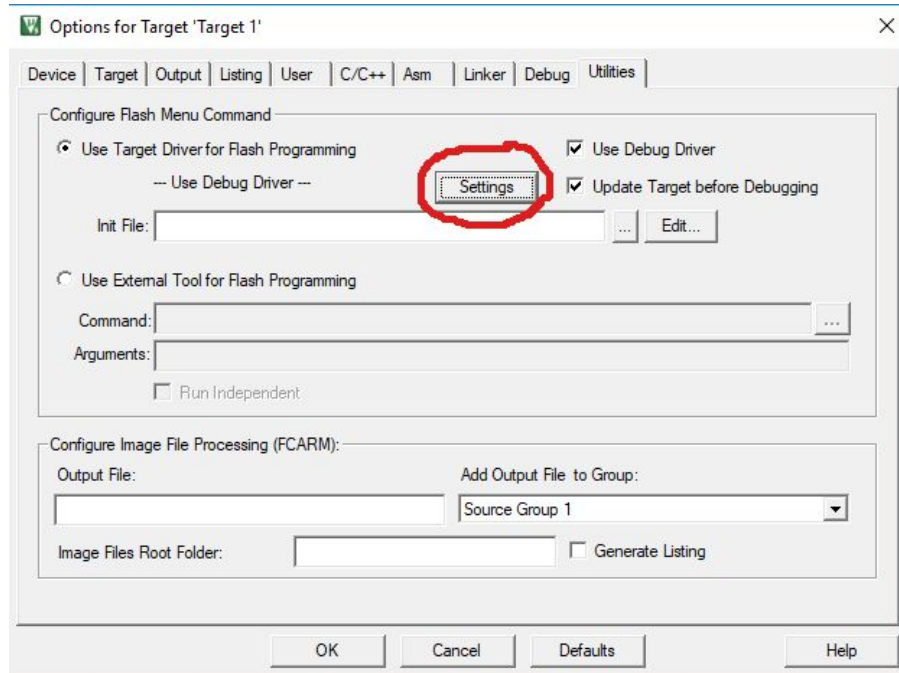


Figure 27: Utilities tab of the "Options for Target..." menu

This will open the Cortex-M Driver Setup window as shown in Figure 28. Notice that the Programming Algorithm section is empty. This is the source of the error. Click Add.

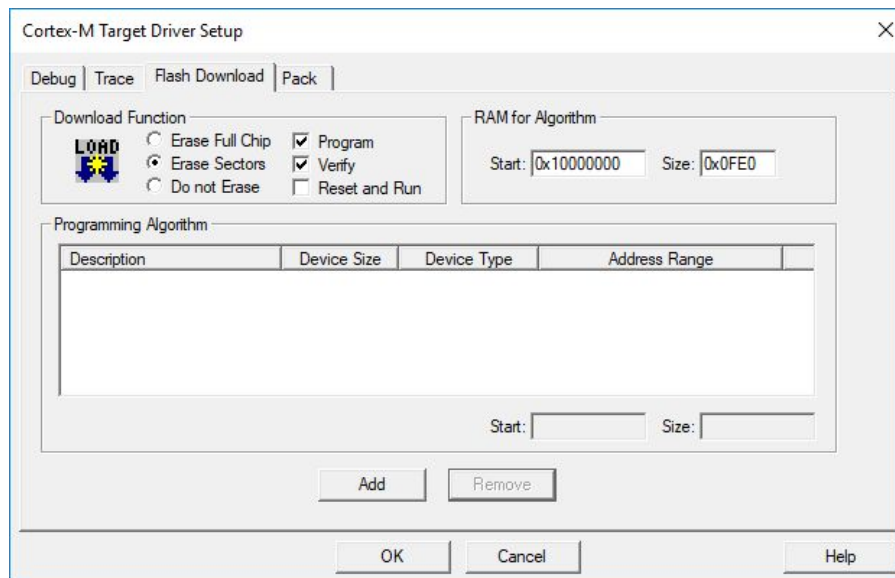


Figure 28: Cortex-M Driver Menu Setup window

Scroll down until you find LPC17xx IAP 512 kB Flash in the list then select it, as shown in Figure 29. Click Add. Click OK twice to close the

other menu windows. You should now be able to load your code to the board.

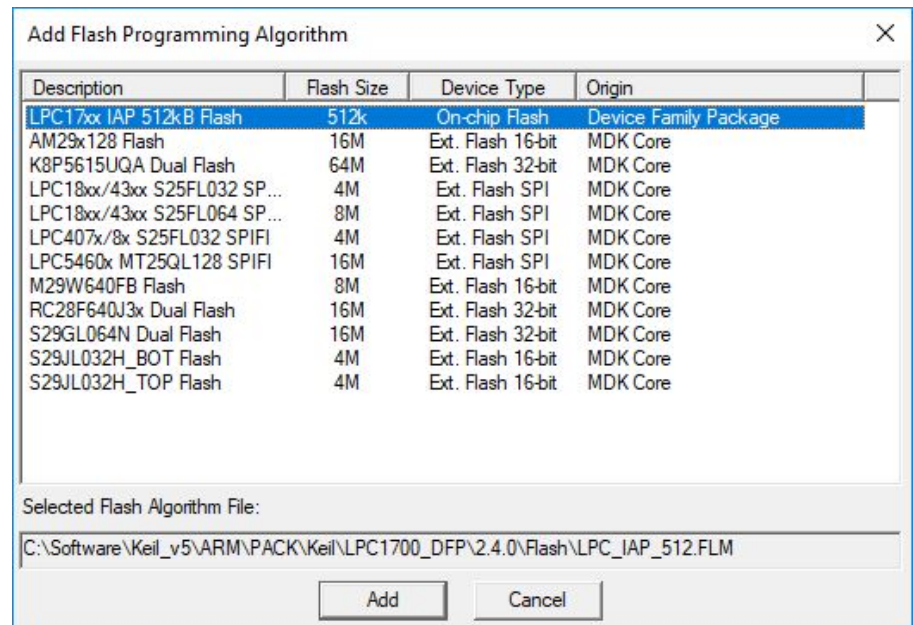


Figure 29: Select the microcontroller on the board to add the correct programming algorithm

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.