



## Lab 1: Compiling, Running, and Debugging

### Introduction

This is the first lab in CSE013E. Here we will demonstrate the basics of compiling and running C programs in the simulator and on the Uno32 hardware. We will also explore the tools we will use and some of their features for debugging problems you might encounter. This lab is shorter than the rest of the lab as it also includes learning Git.

### Reading

- Document on Unix and Git
  - If you're using Windows, see also the document on using WSL
- Document on software installation (if you want to run everything on your own computer)
- Document on style guidelines
- Document on MPLAB X
- Document on compiler errors
- Document on serial communications
- K&R Preface and Introduction
- K&R Sections 1.0-1.2, 4.5, 4.11

### Required Files:

Every lab has a list of required files. These are the files we grade. They must be named EXACTLY as below, including capitalization. Many students mistakenly submit files like "Part1.c", "part2\_template.c", "part3.c.c" etc. Don't do that! Also, there should be exactly one of each of these each LabX folder in your repo. A common mistake is to duplicate some of these files, for example having a "Lab1/README.md" as well as a "Lab1/part0.X/README.md". These issues will disrupt our grading software!

- part0.c
- part1.c
- part2.c
- part3.c
- part4.c (for extra credit)
- README.md

## Lab Files:

Every lab also begins with a list of the files that play a role in the lab, and how we expect you to use them:

- **DO NOT edit** these files:
  - BOARD.c/h - Contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values.
  - Oled.c/h, Ascii.c/h, OledDriver.c/h – These files provides all the code necessary for calling the functions in Oled. You will only need to use the functions in Oled.h, the other files are called from within Oled Library. BOARD.c/h – Standard hardware library for CSE013E.
- **Edit** these files:
  - part1.c, part2.c, part3.c – these contain code you must edit slightly. These files are included in your lab .zip file as partX\_template.c. BE SURE TO RENAME THEM!
    - part1.c: This file contains code that performs a simple sorting algorithm on five randomly generated numbers. Follow the setup procedures listed below, add the requested documentation, and format the code to follow the provided style guidelines.
    - part2.c: This file contains an empty main() to be filled with the exercises from section 1.2 of K&R. In addition, you will be asked to modify these exercises to add some additional functionality. Detailed steps are listed below.
    - part3.c: for extra credit, modify this file to print text on the the IO board's OLED screen
- **Create** these files:
  - part0.c – the main file for part 0, containing code to perform a basic “hello world” using both unix and the PIC32. You will fill it verbatim with text we give you below.

## Assignment requirements

0. Perform “Hello World” with the UNO32
1. Complete the requested modifications to this code.
  - Complete the setup procedures
  - Add the requested documentation to a README.md file
  - Format the code to follow the provided style guidelines
2. Complete the temperature conversions tables based on code provided.
  - Implement code to output a table of equivalent Fahrenheit and Celsius values.

- Extend this code to also output a table of equivalent Kelvin and Fahrenheit values.
  - Format the code correctly according to the style guidelines.
3. Extra credit: “Hello World!” on the OLED.
  4. Make final notes in the README document

## Grading

This assignment consists of 10 points:

- Two points for Part 0
- Four points for Part 1
- Four points for Part 2
- One point of extra credit for Part 3

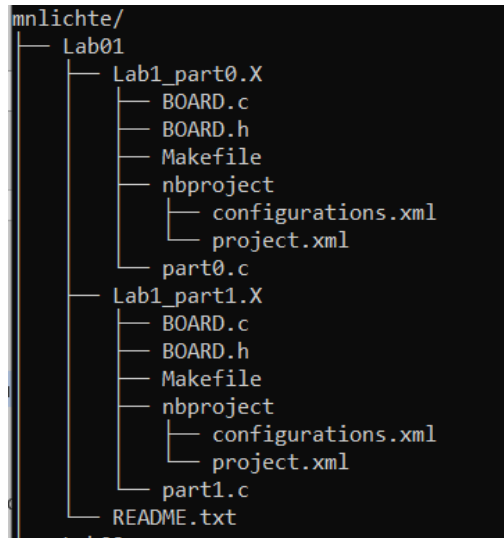
Note that you will lose points for any and all code that fails to compile! All Code for this lab except Part 0.1 must compile within MPLABX regardless of whether your lab kit has arrived or not. This generally means you have at least run it in the simulator.

## A note on git usage and project structure in the time of COVID-19:

Since TAs and tutors cannot work directly on your machine, it is important that we have a mechanism to access your projects in their entirety to assist you in debugging. The tool we use to accomplish this is git. If you ask for help understanding an issue, we will attempt to recreate that issue by pulling your git repo.

For this to work effectively, we need you to carefully adhere to the following rules:

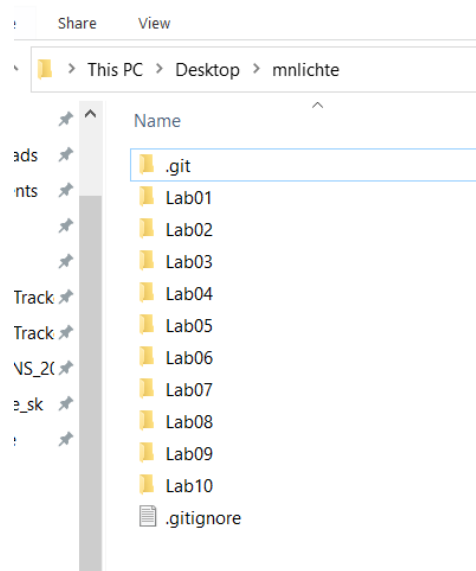
- Commit and push often. Ideally, push every hour or so, and every time you ask for help. It will not take long to become comfortable with this process.
- Keep all of your project files self-contained in the .X folder. Each MPLABX project is stored in a folder whose name ends in .X. You will use a variety of files in each project, including class libraries like Board.h/.c, as well as project-specific files like part1.c. Here is the structure we need:



- Make sure that you commit all of your project files, including project properties and makefiles! There are three crucial files in each project that we need you to commit that you don't work with directly. If you commit and push these, we will be able to access your project properties:
  - \*.X/nbproject/configurations.xml
  - \*.X/nbproject/project.xml
  - \*.X/Makefile

## First Step: Pushing with Git

Our very first step will be establishing our basic ability to commit and push with git (a sort of git “Hello World”). Please use the Git document on CANVAS to clone your repo using Git Bash on Windows or your native terminal on other systems. Be sure to clone your Repo in a place that makes sense (the ‘pwd’ command can be useful here). Once you have cloned your repo verify that you can view the repo with your system’s file explorer:



Now, we will push some changes:

- Make a change to your Lab01 README .md using Notepad or a text editor of your choice.
- Then, using the command line, you should be able to view the change with `$git status`, stage the change with `$git add`, commit the change with `$git commit`, and push the change to the server with `$git push` (again, the “git and UNIX” document will help you understand what is happening, and troubleshoot issues). Here’s what that should look like:

```
mnlichte@LAPTOP-JR1C04A4: /mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.txt

no changes added to commit (use "git add" and/or "git commit -a")
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git add -A
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt

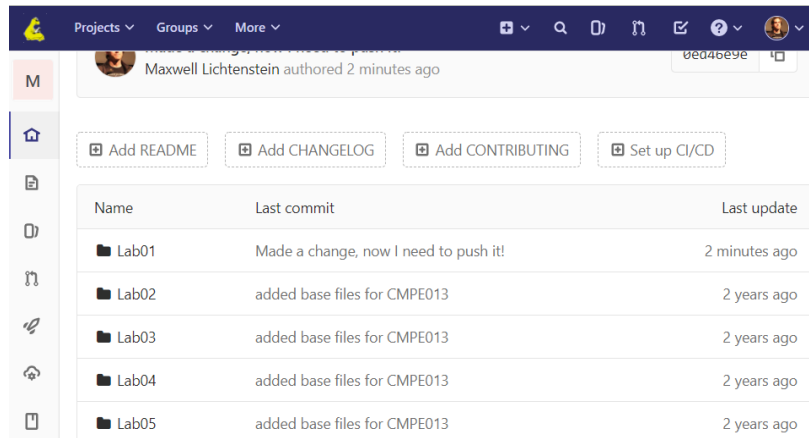
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git commit -m "Made a change, now I need to
push it!"
[master f6ccc5b] Made a change, now I need to push it!
1 file changed, 10 insertions(+), 1 deletion(-)
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git push
Username for 'https://gitlab.soe.ucsc.edu': mnlichte
Password for 'https://mnlichte@gitlab.soe.ucsc.edu':
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 469 bytes | 42.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To https://gitlab.soe.ucsc.edu/gitlab/cse13e/spring20/mnlichte.git
   0ed46e9..f6ccc5b  master -> master
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
mnlichte@LAPTOP-JR1C04A4:/mnt/c/Users/mnlic/OneDrive/Desktop/mnlichte/Lab01$
```

Note that it is helpful to frequently type `$git status` to see what is happening!

The best test of a successful push is seeing the changes appear via the gitlab web portal. If you can see a change on your web portal, then we can see it too! You should see something like the following on <https://gitlab.soe.ucsc.edu/gitlab/cse13e/Quarter/yourcruzid> (this is not a valid link)



If you can see that, you've pushed successfully!

DO THIS OFTEN. Push your work at least every hour of work, and each time you ask for debugging help from staff.

## Part 0 – Hello World

### Part 0.0 – “Hello World”, embedded


All programming languages have a “hello world” program<sup>1</sup>. This program is generally simple as possible and demonstrates that both the code and system runs. While the “Hello World” performed on the UNO32 prints “Hello World,” other embedded systems might have a blinking LED to indicate success.

1. Follow the MPLAB X new project instructions using the “New MPLAB X Project” document on Canvas to generate a project for this part. Please be sure to work in your git repo so the graders can assist you<sup>2</sup>.
2. Right-click on source files and click on New->Other...
  - a. Choose “C -> C Main File” and click “Next.”
  - b. Name the file part0.c and click “Finish”. This will generate a new main file for the project.

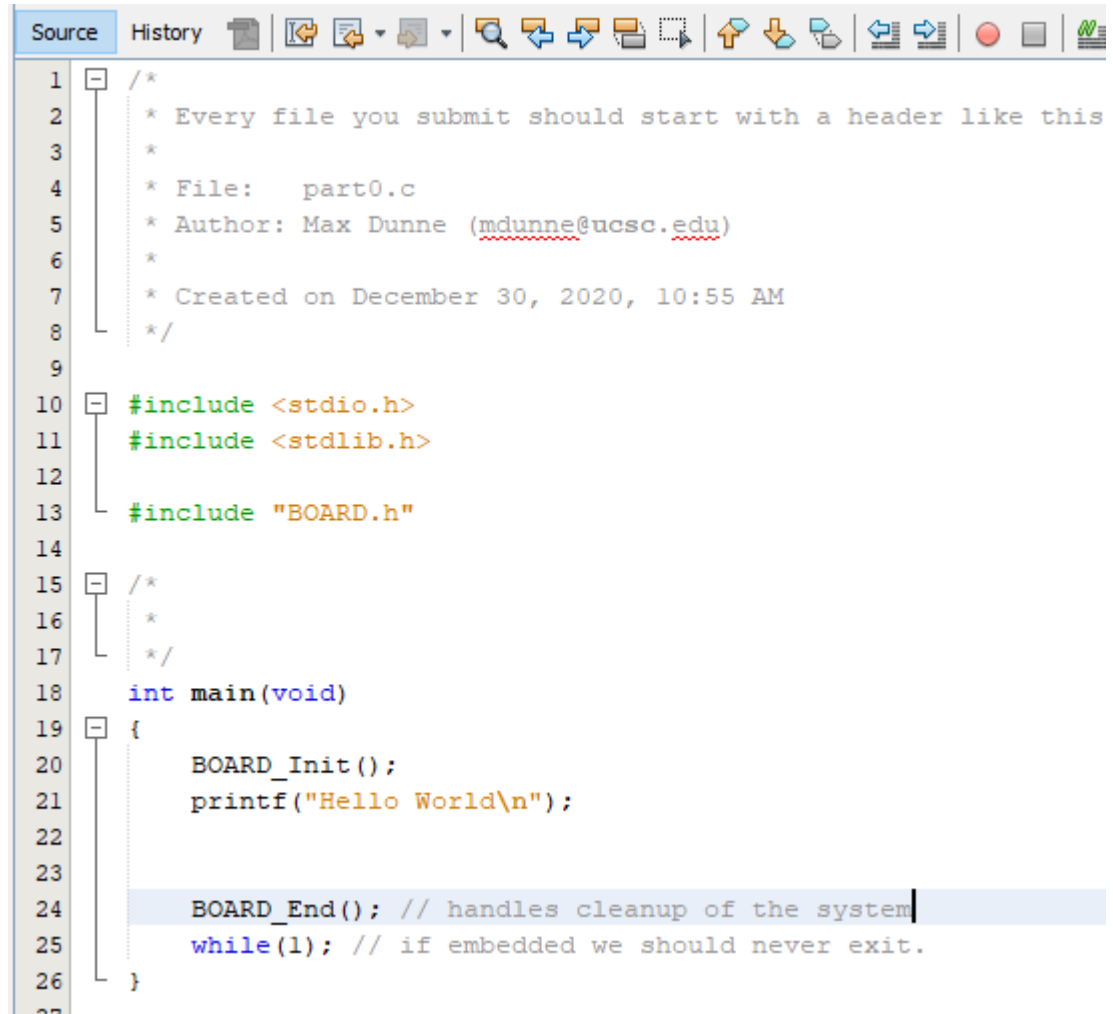
<sup>1</sup> In fact, this is true for all languages because K&R states so at the beginning of chapter 1: “The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: Print the words

*hello, world*

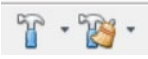
This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.”

<sup>2</sup>  This symbol designates a good place to commit in the process. This should get you used to committing early and often.

- i. Remember for future reference that only one file with a main() function can be added to your project at a time.
- c. Make the contents of this file appear exactly as shown below (except change the names and dates!)

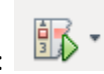


```
1  /*
2   * Every file you submit should start with a header like this
3   *
4   * File:    part0.c
5   * Author:  Max Dunne (mdunne@ucsc.edu)
6   *
7   * Created on December 30, 2020, 10:55 AM
8   */
9
10 #include <stdio.h>
11 #include <stdlib.h>
12
13 #include "BOARD.h"
14
15 /*
16  *
17  */
18 int main(void)
19 {
20     BOARD_Init();
21     printf("Hello World\n");
22
23
24     BOARD_End(); // handles cleanup of the system
25     while(1); // if embedded we should never exit.
26 }
27
```

- 3. Now press one of the “build” buttons:  . This will run the compiler. Note that the “clean build button” takes longer, but sometimes produces more useful error messages.
  - a. If this fails for some reason and your code looks exactly the same as shown, READ THE ERROR MESSAGES. Often, the pertinent error message will tell you what is wrong, and sometimes will even tell you what you need to fix.
  - b. If you don’t understand the errors, make sure you go back and check the new project document.
  - c. If it still does not work and you are sure you have followed the new project instructions correctly, get help from the teaching staff.
- 4. At the bottom of the window will be an Output tab. If the build was successful, you will see a green “Build Successful” message.




- a. You may also see some warnings – if so, figure out where they are coming from and fix them right away! Warnings are there for a reason.



5. Now we must read the output. Run the debugger with the debug button:

- a. You may need to enable UART1 output, by going to “Project Properties -> simulator -> UART1 IO Options -> Enable Uart1 IO”
- b. The output should open up several tabs, including “One of the Output tab’s subtabs will be title "UART 1 Output", selecting this will show the serial terminal output. It should show the words “Hello World” showing that we have successfully ran the hello world program.

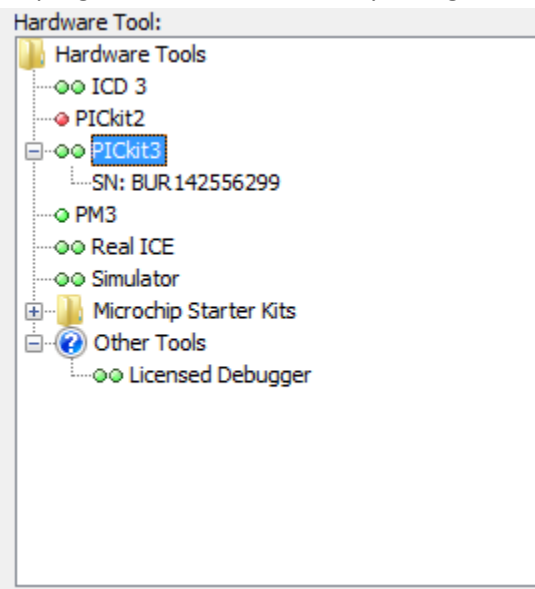


6. Click on  to stop the debugger and continue working.

7. Now that we have successfully run the code on the simulator we can run the code on the UNO32 itself. This process is started by plugging in both USB cables that came with your kit into the computer.


8. Once the drivers have installed themselves go back into the project properties.

- a. To program the UNO32 the only change needed is to select PICkit3 as shown below:



- b. Hit “OK” and exit out of the project properties. Don’t worry about the SN as each PICkit3 has a unique one.



- c. We can now hit the Make and Program Button (  ) to load the code onto the UNO32. To actually see the output you will need to setup a serial port as described in the serial communications document.
- d. Once the serial port is setup you can hit the program button and MPLAB-X will load the code onto the UNO32 and run it.
  - i. If you encounter any errors, recheck the serial port documentation.
  - ii. Note that sometimes there are issues with the PICkit3 connecting.

- iii. If it is still not working, contact the teaching staff.
- e. You should now see “Hello World!” on your serial program window.

## Part 0.1 – “Hello World”, OS

Now that we’ve gotten Hello World using the XC32 compiler and the simulator, it’s time to obtain the same result using gcc and your local operating system.

1. Using the command line (Either WSL on Windows, or Terminal on Mac), navigate to your Lab01\_part0.X folder. Use the “\$ cd” command.
  - a. Use the “\$ ls” command. You should see BOARD.c/h, and part0.c
2. Now we will compile your program with gcc.
  - a. If you haven’t already installed gcc and make, do so now (refer to the WSL document on canvas).
  - b. Compile with the following command:

```
$ gcc -S part0.c BOARD.c
```

- c. If successful, type the “ls” command. You should see two new files called “part0.s” and “BOARD.s”. Use the command “\$ cat part0.s” to see the contents. It will appear confusing as it is not an assembly language you are familiar with
  - d. Now assemble the code with the following command:

```
$ gcc -c part0.s BOARD.s
```

- e. If successful, type the “ls” command. You should see two new files called “part0.o” and “BOARD.o”.
  - f. Finally. Link the code with the following command:

```
$ gcc part0.o BOARD.o
```

- g. There should now be a file called “a.out”. This is called a *binary*, and it contains machine instructions that your local computer can execute.
3. Finally, run the compiled program with:


```
$ ./a.out
```

- a. You should see “Hello World” appear on your terminal.
4. Generally though we do not generate assembly files first but compile to machine code directly. The command below will do this and we will augment this later when we start to work with more files.




```
$ gcc part0.c BOARD.c
```

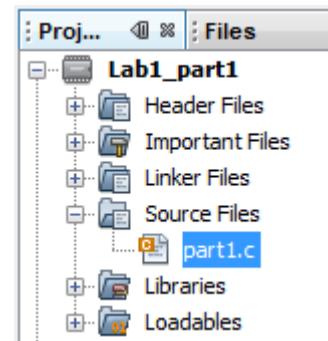
## Part 0.2 – committing


This is an excellent time to git commit and push! You already knew this from the  icon, but just to make sure, we'll walk through it one more time:

- Check your changes with “`$ git status`”
- Stage the appropriate files, for example with “`$ git status *.c *.h *.xml *Makefile README.md`”
- Check that they're staged with “`$ git status`”
- Commit with something like “`$ git commit -m "Committed Lab1 part0 files!"`”
- Push with “`$ git push`”
- Check the push on the gitlab web portal

## Part 1 – Debugging and Code Style

1. **No hardware needs to be connected yet.**
2. Create a new project in MPLAB X using the new project guide.
  - a. Be sure to select the simulator.
  - b. Name the project something that makes sense like “Lab1\_part1”.
  - c. Copy the BOARD.c/h files into the Lab1\_part1.X folder, along with part1\_template.c.
    - i. Don't forget to remove the “\_template”!
3. Go Into project properties and navigate to simulator->Uart1 IO Options and check Enable Uart1 IO as was done in the previous section.
4. Add part1.c to the project (Right-click on the "Source Files" folder in the project window obtainable from View -> Project) 
5. Build the project by clicking the hammer on the toolbar.
  - a. You should see black “BUILD SUCCESSFUL” text at the end of a successful build. A failed build will show a red “BUILD FAILED” message. Get help if your build fails for part 1.



6. Now press the “Debug Main Project” button: . This will start the simulator.

- a. Debug controls, in order: Debug, Stop, Pause, Reset, Continue



7. Press the pause button. It should be located to the right of the “Debug Main Project” button. The debugger should be stopped on the final `while(1);` at the end of the code. A green bar with an arrow on the side is used to indicate the line of code the debugger has paused at (this line of code has not been executed yet).

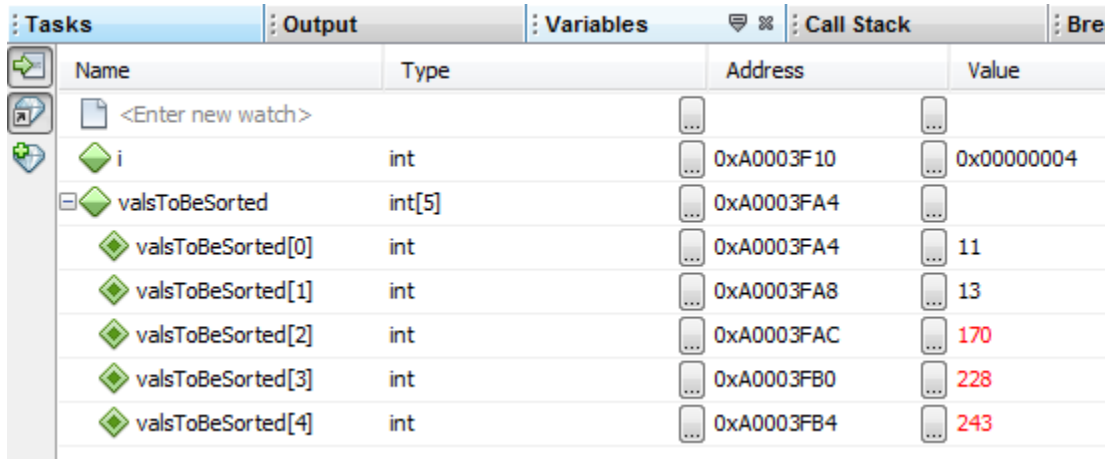
```
72 |  
73 | /*  
74 |  * Returning from main() is bad form in embedded environments. So we  
75 |  * sit and spin.  
76 |  */  
77 | while (1);  
   | }
```

8. At the bottom of the window will be an Output tab. One of its subtabs will be titled "UART 1 Output", selecting this will show the serial terminal output. It should be a sorted list of five numbers, like the following: "[46, 92, 105, 174, 212]".
9. Press the reset button in MPLAB (the blue one with two arrows). This will reset the debugger to the start of `main()` and stay paused. Don't press play quite yet.
10. Place a breakpoint on the first line inside the curly braces of the first for-loop (line 85). Do this by clicking the line number in the left margins of the source code view. It should place a red square on the line number and highlight the whole line red. Placing the breakpoint inside the for-loop ensures that the debugger will stop before every iteration of the loop. Note that you may only have 4 breakpoints active at a given time.

```
81 | // Sort the array in place.  
82 | int i, j;  
83 | for (i = 0; i < 5; ++i)  
84 | {  
85 |     int aTemp = valsToBeSorted[i];  
86 |     for (j = i - 1; j >= 0; j--) {  
87 |         if (valsToBeSorted[j] >
```

- a. Removing a breakpoint is done by pressing the red square again.

11. Now press play. The debugger should pause at the top of the for-loop, changing that line to green and showing a green arrow in the left margin.
12. Below the code window should be a number of tabs. Click the one labeled “Variables”. This tab shows the values of all variables within the scope of where you are paused. It will be updated whenever the program is paused, but not when it is running.
13. Find the array `valsToBeSorted`. You can expand it to see the value of every element of the array by clicking the plus-sign/arrow next to it.

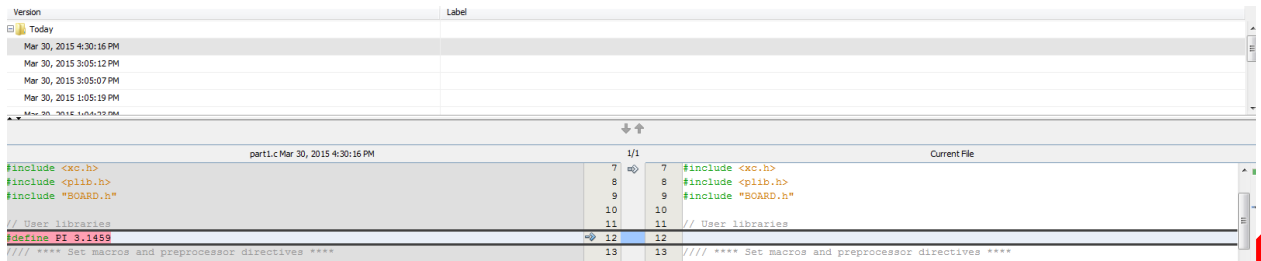


The screenshot shows the 'Variables' tab in a debugger. It contains a table with columns: Name, Type, Address, and Value. The variable `i` is of type `int` at address `0xA0003F10` with value `0x00000004`. The array `valsToBeSorted` is of type `int[5]` at address `0xA0003FA4`. It is expanded to show its elements:

Name	Type	Address	Value
<code>&lt;Enter new watch&gt;</code>			
<code>i</code>	<code>int</code>	<code>0xA0003F10</code>	<code>0x00000004</code>
<code>valsToBeSorted</code>	<code>int[5]</code>	<code>0xA0003FA4</code>	
<code>valsToBeSorted[0]</code>	<code>int</code>	<code>0xA0003FA4</code>	<code>11</code>
<code>valsToBeSorted[1]</code>	<code>int</code>	<code>0xA0003FA8</code>	<code>13</code>
<code>valsToBeSorted[2]</code>	<code>int</code>	<code>0xA0003FAC</code>	<code>170</code>
<code>valsToBeSorted[3]</code>	<code>int</code>	<code>0xA0003FB0</code>	<code>228</code>
<code>valsToBeSorted[4]</code>	<code>int</code>	<code>0xA0003FB4</code>	<code>243</code>

14. Right-click on the `valsToBeSorted` and select “Display Value Column As” -> “Decimal” to see everything as base-10.
15. Record the values of the elements in `valsToBeSorted` in a new file, `README.md`.
  - a. Also place your name at the top of the `README.md`, along with the names of anyone you collaborated with.
16. Press play. It should stop at the top of the for-loop again. Record the values of `valsToBeSorted` again in your `README.md` file and repeat 4 more times (the breakpoint will not be hit on the last run).
17. After the last run through the loop it will continue to the end of the program and will not hit the breakpoint we placed. The final sorted result of the array will be shown in the Output tab under the UART 1 Output sub-tab.
18. **Complete Task 1 - Add documentation**  
 Add a comment directly above the sorting for-loop, but below the comment that says "Sort the array in place", listing the 5 recorded values of `valsToBeSorted` from the last run (some may be the same, but list them anyways).
19. **Complete Task 2 - Correct formatting**

- a. part1.c does not conform to the dictated style guidelines. While we could fix the source code manually MPLAB X has the capability to reformat the code for us. Press “Alt-Shift-F” and watch as MPLAB-X fixes all the code for you.
- b. After saving the properly formatted Part1.c we can view the history of this file by right-clicking on the filename in the bar and selecting Local History -> Show Local History. This brings up the interface as shown below.



- c. Clicking on any of these dates will compare the current file to its content when it was saved. (Note that this history is local to the computer and by default only lasts 7 days)
- d. Similarly if instead of selecting Show Local History we click on Diff To... we can compare it to an arbitrary file. In this case we can compare it to the Part1.c before we started by using a fresh copy of the file.
- e. Open up this diff and record the line numbers where changes were made to the code, not the comments, to your README.md.

## Part 2 – Temperature Conversion

### 1. Standard Conversion Table

- a. Create a new MPLAB project named Lab1\_part2 following the same procedure as in Part 1
  - i. Don't forget to enable the additional warnings in the compiler options!
  - ii. Make a new subfolder to keep things nicely organized.
- b. Add the provided "part2.c" in it to your new project
- c. Implement the code example below<sup>3</sup>. Type out the code between the comments that say “Your code goes in between this comment and . . .” in your part2.c file.

<sup>3</sup> Note that this is example is from K&R 1.2, please reread this section before attempting part 2 of the lab.

```
// Declare Variables
float fahr,celsius;
int lower,upper,step;

// Initialize Variables
lower=0;    // lower limit of temperature
upper=300;  // upper limit
step=20;    // step size
fahr=lower;

// Print out table
while(fahr<=upper){
    celsius=(5.0/9.0)*(fahr-32.0);
    printf("%f %f\n", (double)fahr, (double)celsius);
    fahr=fahr+step;
}
```

- d. Compile and run your code in the simulator by clicking the debug button that you used for part 1. Now check that your output looks similar to the following output:

```
0.000000 -17.777778
20.000000 -6.666667
40.000000 4.444445
60.000000 15.555557
80.000000 26.666667
100.000000 37.777778
120.000000 48.888893
140.000000 60.000003
160.000000 71.111114
180.000000 82.222229
200.000000 93.333335
220.000000 104.444450
240.000000 115.555557
260.000000 126.666671
280.000000 137.777786
300.000000 148.888900
```

- e. Fix the coding style for this code such that it follows the provided Style Guidelines. To do this, select all of the code and click Source -> Format (or alt-shift-F) to allow MPLAB X to automatically fix the formatting. Note the differences between the formatted and unformatted code (use Edit -> Undo and Edit -> Redo to compare or use the local history as in part 1).
- f. Change the display format (currently "%f") of the Fahrenheit value so that it has a minimum width of 7 characters and a precision of 1. Also change the display format of the Celsius value so that it has a minimum width of 4 characters, it's left-padded with 0s, and displays no characters after the decimal point.

- i. To see an explanation of these format specifiers, click on `printf()` in your code, and press CTRL+SPACE to bring up Code Assistance, and scroll down to the bullet points. To see examples, scroll all the way to the bottom. Code assistance can also be used for auto-completion, which we will later see.

The screenshot shows the MPLAB X Code Assistance window for the `printf()` function. The top part shows the function signature: `printf(const char*, ...) int`. Below this, there is a detailed explanation of the format string, followed by a list of flags and their descriptions. The flags listed are: `'#'` (alternate form), `'0'` (padding with zeros), `'x'` (lowercase hexadecimal), `'X'` (uppercase hexadecimal), `'a'`, `'A'`, `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, and `'G'`.

- ii. You can also access this information through help in MPLAB X as well. Go to "Help" -> "Help Contents" -> "XC32 Toolchain" -> "XC32 Standard Libraries" -> "Standard C Libraries with Math" -> "<stdio.h> Input and Output" -> "STDIO Functions" and find `printf` (note that we think ctrl-space is simpler).

- g. The output should now look as follows:

```
0.0 -018
20.0 -007
40.0 0004
60.0 0016
80.0 0027
100.0 0038
120.0 0049
140.0 0060
160.0 0071
180.0 0082
200.0 0093
220.0 0104
240.0 0116
260.0 0127
280.0 0138
300.0 0149
```

F	C
0.0	-018
20.0	-007
40.0	0004
60.0	0016
80.0	0027
100.0	0038
120.0	0049
140.0	0060
160.0	0071
180.0	0082
200.0	0093
220.0	0104
240.0	0116
260.0	0127
280.0	0138
300.0	0149

K	F
0.000	-459.669982
20.000	-423.669982
40.000	-387.669982
60.000	-351.669982
80.000	-315.669982
100.000	-279.669982
120.000	-243.669982
140.000	-207.669982
160.000	-171.669982
180.000	-135.669982
200.000	-99.669982
220.000	-63.669982
240.000	-27.669986
260.000	8.330012
280.000	44.330009
300.000	80.330009

## 2. Column Headers



- a. Add column headers to above the Fahrenheit-to-Celsius table with the letters 'F' and 'C' spaced nicely using the `printf()` function you read about in the MPLAB X help. This header should be printed only once and be before any of the conversions are calculated and printed.

### 3. Kelvin to Fahrenheit Table

- a. Now print a newline character after the Fahrenheit-to-Celsius table by typing `prin`, press `CTRL+SPACE`, and then press `ENTER` to auto-complete<sup>4</sup> into `printf()`. Use the string `"\n"` to add a blank line in the output.
- b. Now make a Kelvin to Fahrenheit conversion table, complete with its own header. This can be done by selecting the block of code that you modified:

```
fahr=lower;

// Print out table
while(fahr<=upper){
    celsius=(5.0/9.0)*(fahr-32.0);
    printf("%f %f\n", (double)fahr, (double)celsius);
    fahr=fahr+step;
}
```

Then copy (Edit -> Copy), and paste (Edit -> Paste) the code below the `printf()` you just added.

- i. Rename the variable `fahr` by selecting the pasted block of code, and click Edit -> Replace. For "Find What:" type `fahr`, and for "Replace With:" type `kelv`. Click "Replace All", and Close.
- ii. Notice how the `kelv` variable was not defined (MPLAB X underlines it in red). To fix this, declare it as a float by adding `float` in front of `"kelv=lower;"`.
- iii. Now replace `celsius` with `fahr` in the same block of code. Next, modify the line `"fahr = ..."` by changing the expression to convert from kelvin to Fahrenheit.
- iv. Now format the table to look like the image on the previous page. The Kelvin values should be displayed in the left-hand column with a minimum width of 3 characters, left-padding with zeros, and a precision of 3. The Fahrenheit values should be displayed in the right-hand column with a minimum width of 5 characters.

---

<sup>4</sup> Auto-complete, that is typing the first few characters and `ctrl-space` is your friend; this will complete function names, variables, members of structs, `#defines`, pretty much anything defined in scope in the current project. It is incredibly useful, use it often.

- c. Note that a compiler error will occur if two variables of the same name are declared. You need to have unique names for all variables within a single scope (scope will be explained later, but for this part of the lab you will need unique names for all variables you use). You can reuse your variables from the first table for generating the second table, just make sure that the “float fahr...” and “int lower,...” lines only appear once at the top<sup>5</sup>. Also you should make sure that your variable names make sense with the values that they store<sup>6</sup>; for instance, you should use a variable named `kelv` to store the calculated Kelvin values.
- d. Refactoring (renaming) variables and functions is another useful feature in MPLAB X. You decide that `kelv` should be renamed to `kelvin` in your code. Click on `kelv` anywhere in your code, and click Refactor -> Rename, and type `kelvin`. **This renames the variable everywhere in the project!** Be careful when using refactor.
- e. Again view the output in the UART 1 Output tab of the Output window. The final output of the program should look like the picture above.

### Part 3 – “Hello World!” on the OLED (extra credit)

You are going to print “Hello, World!” to the OLED on your physical UNO32 board.

In order to do this, you will need to add `ascii.c/h`, `Oled.c/h`, `OledDriver.c/h` to your basic HelloWorld program. In order to use the OLED display, you will need to call the functions: `OledInit()`, `OledDrawString()`, and `OledUpdate()`. Take a look at the `Oled.h` comments to see how to use these functions.

It is possible to complete this if you do not yet have your physical lab kit, but you will be unable to test your work.

### README.md:

---

<sup>5</sup> In general, putting all of your variables at the top of your function is a very good practice.

<sup>6</sup> Good naming of variables and functions is a key part of software design, start practicing it now.



Above, we have asked you to record various results in your README. In addition, please add some thoughts of your own:

- What did you do in this lab?
- What mistakes did you make, and how did you identify and solve them?
- Do you have any feedback for us about this lab?

Make sure that this document is readable and aesthetically pleasing. Your READMEs must all be in markdown. While we do not need you to use advanced markdown formatting they should render correctly within the gitlab interface. When viewing your repository on gitlab any directory that has a README.md file within it will automatically be rendered. More information about the formatting language can be found at <https://daringfireball.net/projects/markdown/syntax> and <https://docs.gitlab.com/ee/user/markdown.html>.

## Frequently Asked Questions:

***I see Connection Failed. in the PICkit3 tab when trying to program the ChipKIT.***

Unplug the PICkit3 and plug it back in. Try to program or debug again. A window will pop up prompting you to reselect the PICkit3 device. If this doesn't work repeat a few more times. If that fails, ask a TA/tutor.

***I see The programmer could not be started: Could not connect to tool hardware: PICkit3PlatformTool, com.microchip.mplab.mdbcore.PICKit3Tool.PICkit3DbgToolManager in the PICkit3 tab when trying to program the ChipKIT.***

Just disconnect and reconnect the USB cable for the PICkit3 and try again.

***How do I know which files to #include?***

A source file (.c) should include its complementary header file (.h). The header file should include anything that's required for it to compile. That is, any types or enums that are used in function declarations should be included in the header. Because the source file includes the header, anything included in the header is effectively already included in the source file. If the source file uses any outside functions or standard library functions, the matching header files for those need to be included as well. You should never include a .c file, only a .h.

Steps should be:

1. Try to compile

2. If a function is not defined (causing an error) find the header file that defines it and include that header.
3. Repeat until there are no undefined functions.

***My code doesn't compile!***

If the error message says something about multiple definitions of 'main' it's because you included two files that have a main function in your project. Each of the three parts of this lab need to be in separate projects; part1.c part2.c and part3.c cannot be in the same MPLAB X project.

If the error message says something about expecting a character or identifier it's probably because of a syntax mistake in the code. You can click on errors to go to where they appear in the code. Keep in mind that for many syntax errors, the error that MPLAB X tells you about may have been caused by an earlier line, and that a single-character mistake can create many errors. Refer to the Compiler Errors document provided for some assistance with the more common compilation errors you will encounter..

***I see Failed to get Device ID in the PICkit3 tab when trying to program the ChipKIT.***

Just disconnect and reconnect the USB cable for the PICkit3 and try again.