UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE13/L: INTRODUCTION TO PROGRAMMING IN C

Lab 1: Compiling, Running, and Debugging

Introduction

This is the first lab in CMPE13. 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.

Reading

- Document on compiler errors
- Document on Unix and Git
- Document on software installation (if you want to run everything on your own computer)
- Document on style guidelines
- Document on MPLAB X
- Document on serial communications
- K&R Preface and Introduction
- K&R Sections 1.0-1.2, 4.5, 4.11

Provided Files

- 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.

- 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. **You will not be modifying these files at all!**

- 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. **You will not be modifying these files at all!**

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.txt 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. You will implement a simple module to implement two very simple functions. Detailed steps are listed below.

4. Extra credit: “Hello World!” on the OLED.

Grading

This assignment consists of 12 points:

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

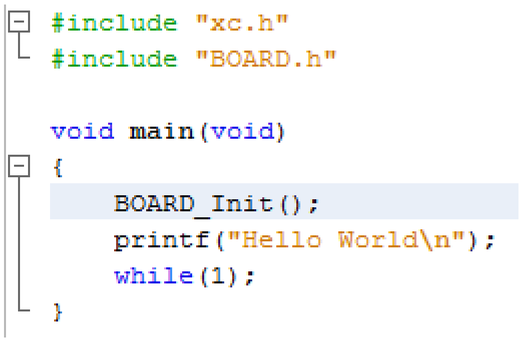
Note that you will lose points for any and all code that fails to compile!

Part 0 – “Hello World”

All programming languages have a “hello world” program[1]. 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] 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

1. Follow the MPLAB X new project instructions (link) to generate a project for this part. I would suggest working with the git repository[2]. We have seeded your repo with folders for each lab. When the graders pull labs this is where they will be pulled from.
2. Right-click on source files and click on New->Other…
    a. Choose Microchip Embedded-> XC16 Compiler and select the mainXC16.c file (We're not using the XC16 compiler, but this will generate the correct file anyway). Click "Next."
    b. Name the file Part0 and click "Finish". This will generate a new main file for the project.
        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 as those shown below



```
#include "xc.h"
#include "BOARD.h"

void main(void)
{
    BOARD_Init();
    printf("Hello World\n");
    while(1);
}
```

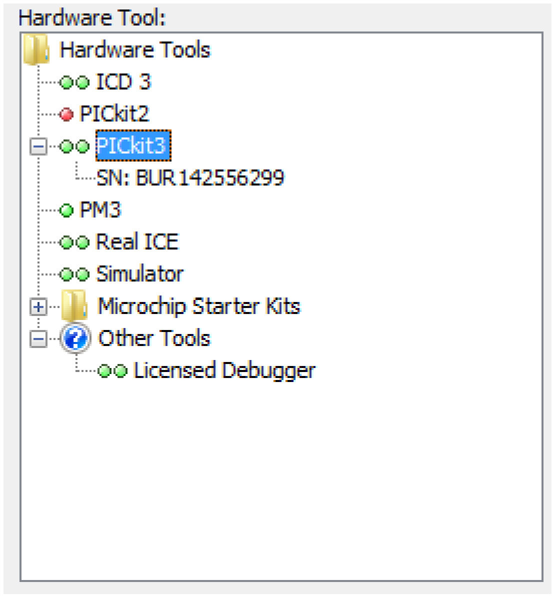
3. Go into project properties and navigate to simulator->Uart1 IO Options and check Enable Uart1 IO.
4. Now press the "Debug Main Project" button: . This will run your code inside the simulator.
    a. If this fails for some reason and your code looks exactly the same as shown, make sure you go back and check the new project document.
    b. If it still does not work and you have followed the new project instructions, get help from the teaching staff.
5. At the bottom of the window will be an Output tab. One of its 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 program 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.

---

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."

[2] This symbol designates a good place to commit in the process. This should get you used to commiting early and often.

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:



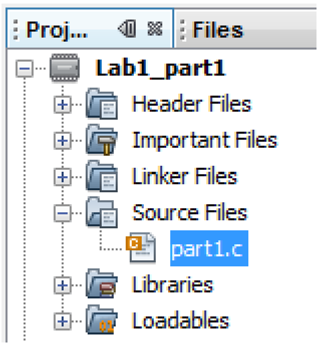
Hardware Tool:

```
Hardware Tools
├─ ICD 3
├─ PICkit2
├─ PICkit3
│   └─ SN: BUR142556299
├─ PM3
├─ Real ICE
├─ Simulator
├─ Microchip Starter Kits
└─ Other Tools
    └─ Licensed Debugger
```

   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 https://classes.soe.ucsc.edu/cmpe013/Spring15/Labs/CMPE13_SerialCommunications.pdf
   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 1 – Debugging and Code Style

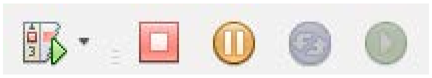
1. **No hardware needs to be connected yet.**
2. Create a new project in MPLAB X using the new project guide being sure to select the simulator. Name the project something that makes sense like “Lab1_part1”.
3. Go Into project properties and navigate to simulator->Uart1 IO Options and check Enable Uart1 IO as was done in the previous



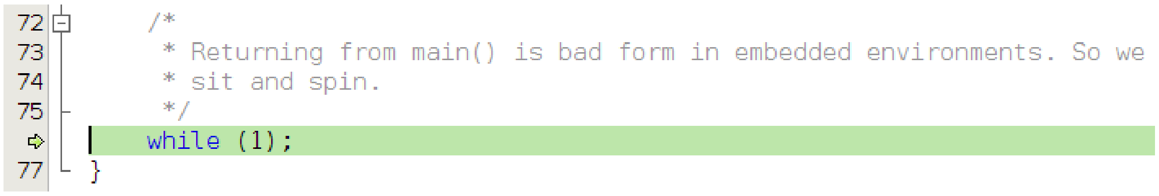
```
Lab1_part1
├─ Header Files
├─ Important Files
├─ Linker Files
├─ Source Files
│   └─ part1.c
├─ Libraries
└─ Loadables
```

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. This will result in a "Lab1_part1.X.production.hex" file under "\Lab1_part1.X\dist\default\production"
   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).



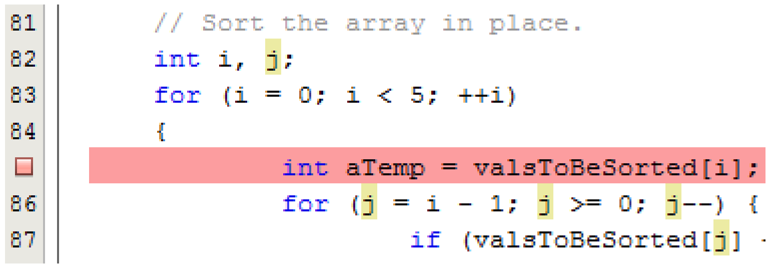
```
    /*
     * Returning from main() is bad form in embedded environments. So we
     * sit and spin.
     */
    while (1);
}
```

8. At the bottom of the window will be an Output tab. One of its subtabs will be title "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

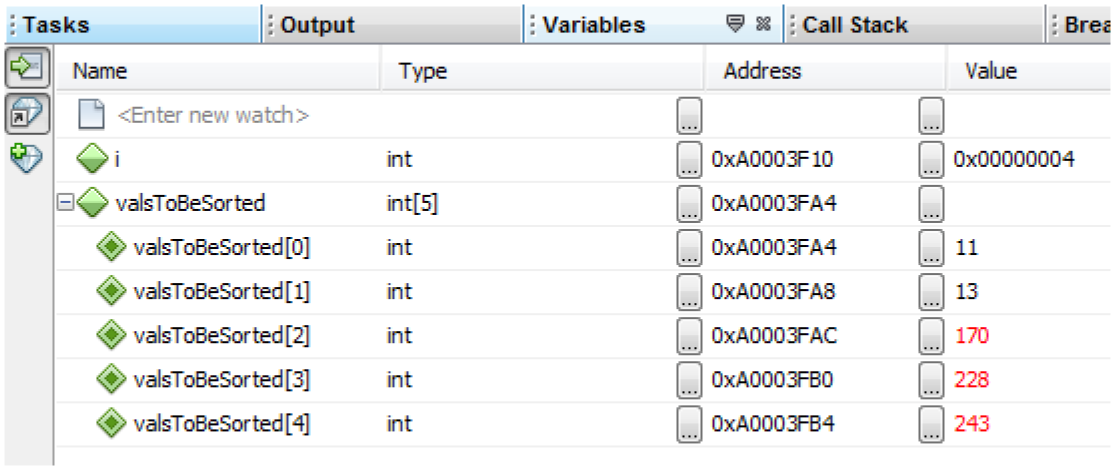
may only have 4 breakpoints active at a given time.



```
    // Sort the array in place.
    int i, j;
    for (i = 0; i < 5; ++i)
    {
        int aTemp = valsToBeSorted[i];
        for (j = i - 1; j >= 0; j--) {
            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.

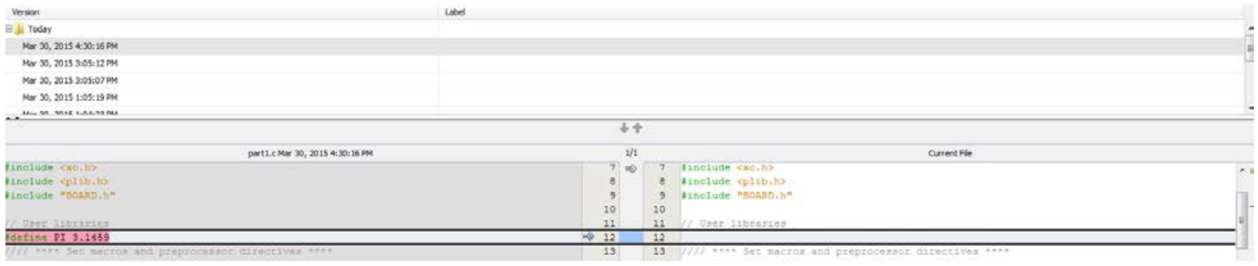


Name	Type	Address	Value
<Enter new watch>			
i	int	0xA0003F10	0x00000004
valsToBeSorted	int[5]	0xA0003FA4	
valsToBeSorted[0]	int	0xA0003FA4	11
valsToBeSorted[1]	int	0xA0003FA8	13
valsToBeSorted[2]	int	0xA0003FAC	170
valsToBeSorted[3]	int	0xA0003FB0	228
valsToBeSorted[4]	int	0xA0003FB4	243

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.txt.
    a. Also place your name at the top of the README.txt, 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.txt 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.



```
#include <xc.h>
#include <plib.h>
#include "BOARD.h"

// User libraries
#define PI 3.1459
//// **** Set macros and preprocessor directives ****
```

```
#include <xc.h>
#include <plib.h>
#include "BOARD.h"

// User libraries

//// **** Set macros and preprocessor directives ****
```

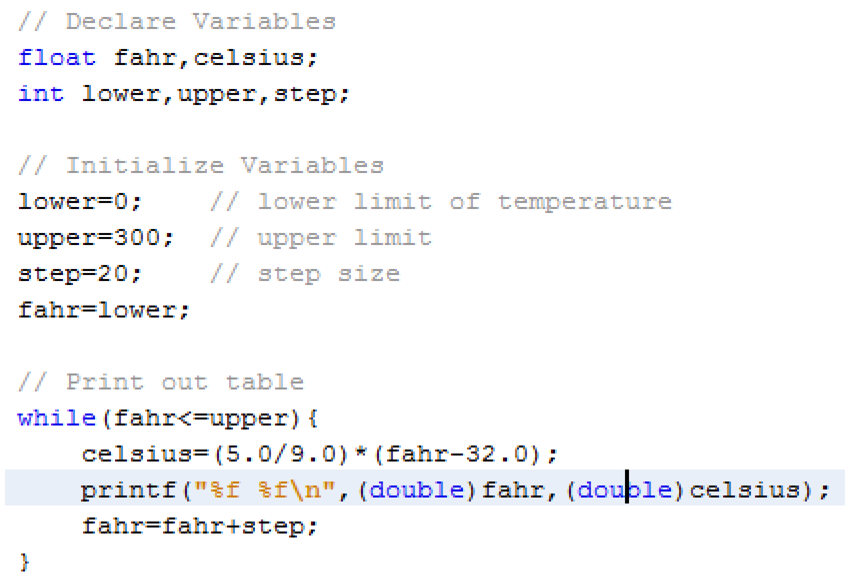
    c. Clicking on any of these dates will compare the current file to its content when it was saved. (Note that this history is local to the computer and by default only lasts 7 days)
    d. Similarly if instead of selecting Show Local History we click on Diff To… we can compare it to an arbitrary file. 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.txt.

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[3]. Type out the code between the comments that say "Your code goes in between this comment and . . ." in your part2.c file.

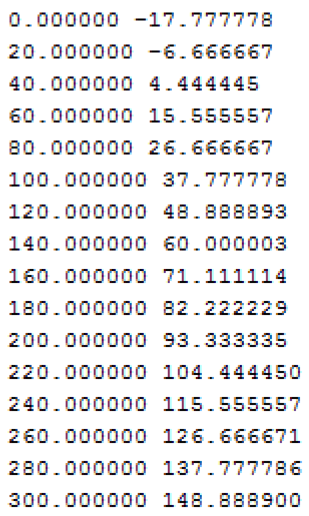


```
// Declare Variables
float fahr,celsius;
int lower,upper,step;

// 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
```

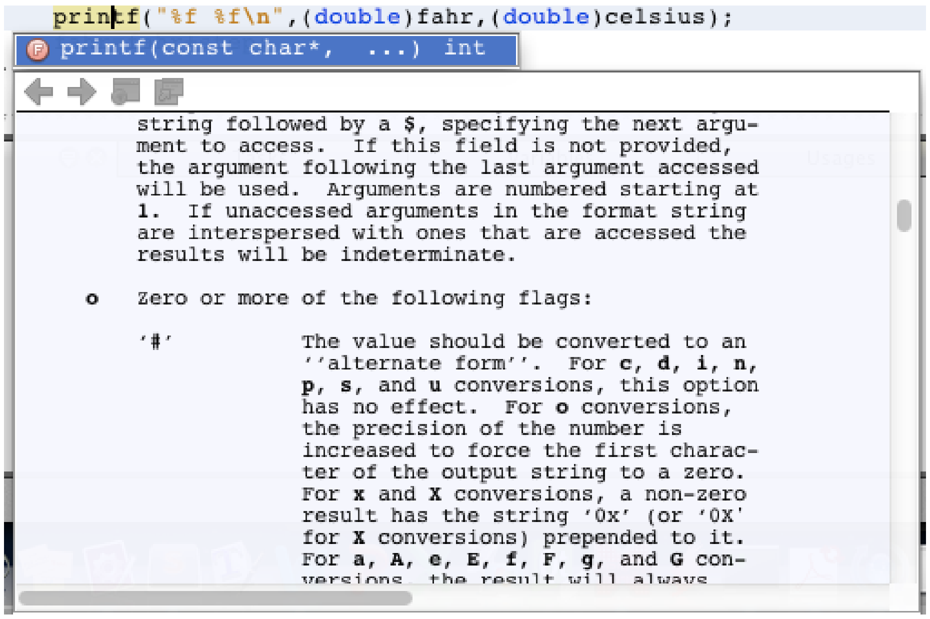
e. Fix the coding style for this code such that it follows the provided Style Guidelines. To do this, select all of the code and click Source -> Format (or alt-shift-F) to allow MPLAB X to automatically fix the formatting. Note the differences between the formatted and unformatted code (use Edit -> Undo and Edit -> Redo to compare or use the local history as in part 1).

[3] Note that this is example is from K&R 1.2, please reread this section before attempting part 2 of the lab.

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.

G

   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.



```
printf("%f %f\n",(double)fahr,(double)celsius);
printf(const char*,  ...) int
```

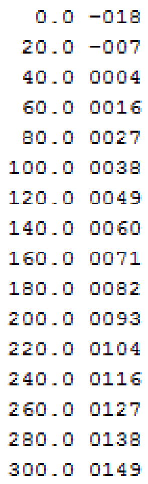
```
             string followed by a $, specifying the next argu-
             ment to access.  If this field is not provided,
             the argument following the last argument accessed
             will be used.  Arguments are numbered starting at
             1.  If unaccessed arguments in the format string
             are interspersed with ones that are accessed the
             results will be indeterminate.

         o   Zero or more of the following flags:

             '#'          The value should be converted to an
                          ''alternate form''.  For c, d, i, n,
                          p, s, and u conversions, this option
                          has no effect.  For o conversions,
                          the precision of the number is
                          increased to force the first charac-
                          ter of the output string to a zero.
                          For x and X conversions, a non-zero
                          result has the string '0x' (or '0X'
                          for X conversions) prepended to it.
                          For a, A, e, E, f, F, g, and G con-
                          versions, the result will always
```

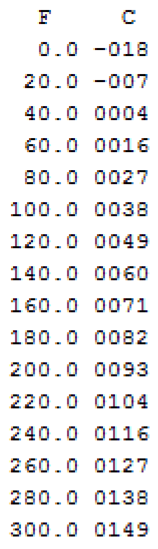
   ii. You can also access this information through help in MPLAB X as well. Go to “Help” ->”Help Contents”->”XC32 Toolchain”->”XC32 Standard Libraries”->”Standard C Libraries with Math”->”<stdio.h> Input and Output”->”STDIO Functions” and find `printf` (note that we think ctrl-space is simpler).

g. The output should now look as follows:



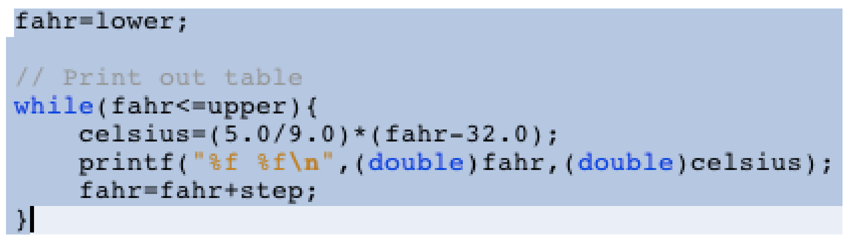
```
  0.0 -018
 20.0 -007
 40.0 0004
 60.0 0016
 80.0 0027
100.0 0038
120.0 0049
140.0 0060
160.0 0071
180.0 0082
200.0 0093
220.0 0104
240.0 0116
260.0 0127
280.0 0138
300.0 0149
```

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.



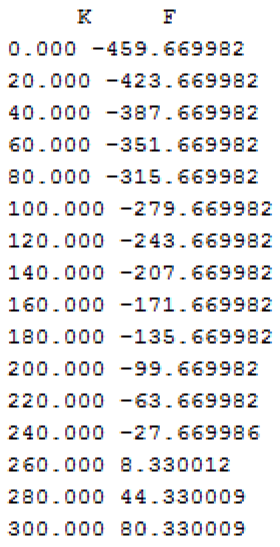
```
    F      C
  0.0 -018
 20.0 -007
 40.0 0004
 60.0 0016
 80.0 0027
100.0 0038
120.0 0049
140.0 0060
160.0 0071
180.0 0082
200.0 0093
220.0 0104
240.0 0116
260.0 0127
280.0 0138
300.0 0149
```

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[4] 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;
}
```



```
     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
```

[4] Auto-complete, that is typing the first few characters and ctrl-space is you 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.

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.

G

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[5]. Also you should make sure that your variable names make sense with the values that they store[6]; 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.

G

[5] In general, putting all of your variables at the top of your function is a very good practice.
[6] Good naming of variables and functions is a key part of software design, start practicing it now.

Part 3 - #include directives and Modules

In this section you are going to learn to build your first module (which is a .c and a .h file that each share the same name) in order to implement two very simple mathematical functions. Note that these are really simple functions that you would never implement this way, but serve to get you used to using functions and how to call them.

The two functions are: IncrementByOne and AddTwoNumbers.

Before we get into the nuts and bolts of implementing these, we need to know what a module in C is. Basically, a module is a pair of files [.c,.h] which are called the "header file," which ends in ".h" and the "source file" which ends in ".c" which are added to a project to encapsulate the functionality of the code into a common holder. The advantage of this form is that is separates how to use the functions (instructions in the .h) versus how the functions are implemented (in the .c). This is particularly useful in complicated projects or where several people are working on the same code, as it enforces partitioning the problem into smaller parts. A useful analogy is that the .h file contains a contract between the end user and the programmer on how to use the functions. As long as the user abides by this contract, the programmer is free to change his/her code without causing the users code to fail.

In order to explore the functionality of a module, we're going to first have you use the built-in math functions of C in order to test the functions, and then later you are going to use your own functions that you write yourself. This example is trivial, but demonstrates how to use functions.

Make a new project with the part3.c file provided, and uses the two math operators + and ++. These correspond to addition and (post) increment. They are used as:

```
z = x + y;
x++;
```

which is to say that the first line adds two numbers, sets a variable with the result. The second line increments a variable. What we are going to want to do is modify the hard coded numbers in there, and run both the addition and increment and make sure you believe the numbers. As a point of interest, keep everything in integer variables.

Once you convince yourself that the math functions are working, and you have a decent test capability, then we are going to substitute our own functions for the ones (in this scenario, we are going to assume that the normal operators are not available).

In order to implement these two functions, we first need to define how the functions are going to work. In the case of the AddTwoNumbers function, we take in two integer values and returns the sum. In order to specify this to C, we will declare a *function prototype* that looks like this:

```
int AddTwoNumbers (int y, int x);
```

For the IncrementByOne, we are going to need only a single integer input, and the function returns a single integer. We are actually going to implement this by using the other function we've created (that is, a function will call another function). Again, to specify this in C, we are declare:

```
int IncrementByOne (int x);
```

Putting this together into a header file, we are going to have a full header file that looks like this:

```
/* module containing implementations of AddTwoNumbers and IncrementByOne */
#ifndef SIMPLEMATH_H
#define SIMPLEMATH_H
int AddTwoNumbers (int y, int x);
int IncrementByOne (int x);
#endif
```

This should go in a file named: SimpleMath.h

In order for your code (in main.c) to use these functions, you will have to include your header file (the source file will get included automatically) by using the C-preprocessor directive: #include "SimpleMath.h"

Note that you will need the #include line inside the SimpleMath.c as well in order for it to compile. This is because C requires that all functions (and variables) be *declared* before they are used.

Now, onto the implementation, which will go into the SimpleMath.c file. Let's tackle the Add Two Numbers first, since it should be more familiar.

We are just going to implement the addition within a function (sometimes very simple functions are known as wrappers or wrapper-functions).

The increment by one function is going to accomplished by calling the add two numbers function with one of the arguments set to 1. That is:

IncrementByOne(x) ↔ AddTwoNumbers(x,1)

If you wanted a helper function, you would put that as a private function within the .c module. This helper function can be used by both of your functions, yet is fully contained within the .c file (thus not exposed to the world). In order for the compiler to be happy, inside the .c file you will need both the *function prototype* and the *function declaration* itself. The function prototype tells the compiler how much space to set aside, and lets you give it the rest of the details (actual code) later. Lets say you needed a function of Bigger, which would return the bigger number. In this case the prototype will look like:

```
int Bigger(int x, int y);
```

This should be up near the top of the .c file. The function itself, is going to appear farther down in the file (lets go ahead and put at the end of the .c file), and is going to look like:

```
int Bigger(int x, int y)
{
	// you get to fill this in
}
```

Note that there are no semicolons at the end of the function. Once you have this function written, you will need to test it. It should work for any two integer numbers, positive and negative number, and also for 0.

Now go ahead and implement the other two functions (IncrementByOne and AddTwoNumbers), and test them both using the previous part3.c main function to convince yourself that they are working correctly. You will need to copy the lines that tested + and ++ and use function calls to your own implementation. This will allow you to compare the results of your own functions directly to what you got from the standard versions and see how close they are. Print out the results from both calculations to see that they are identical.

Part 4 – "Hello World!" on the OLED

You are going to print "Hello, World!" to the OLED on your 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.

Required Files (Case sensitivity matters)

- part0.c
- part1.c
- part2.c
- part3.c (technically optional but nice to have)
- SimpleMath.c
- SimpleMath.h
- part4.c

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 it's 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.