11024 EPC

Embedded C Programming:
Introduction to the C Programming Language

Hands-On Exercises

Table of Contents

Exercises require you to write some code according to the instructions.
Demos are complete applications that you may run to help illustrate particular concepts.

“Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. No representation or warranty is given and no liability is assumed by Microchip Technology Inc. with respect to the accuracy of such information, or infringement of patents arising from any such use of otherwise. Use of Microchip’s products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.”

Lab 1

Variables and Data Types—Demonstration

Purpose

The purpose of this lab is to illustrate how variables are declared, and how the data type of a variable affects the way it is stored in memory. It will also illustrate how to view the variables within MPLAB, both in their high level context as well as machine level context. Upon completion of this exercise, you will understand how to view C level variables in MPLAB and how they are stored in memory based on their data type.

Note, that while we will be using the C30 compiler for these exercises, and that data memory is 16-bits wide, the fundamentals remain the same when used with one of the 8-bit compilers such as MPLAB C18 as well. Only the size of some variables and the width of data memory in the display windows will change.

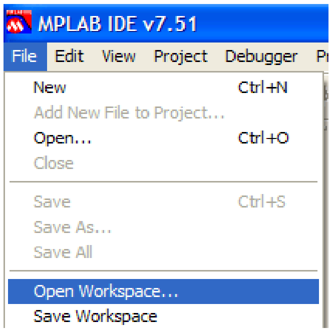
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab01\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab01\...

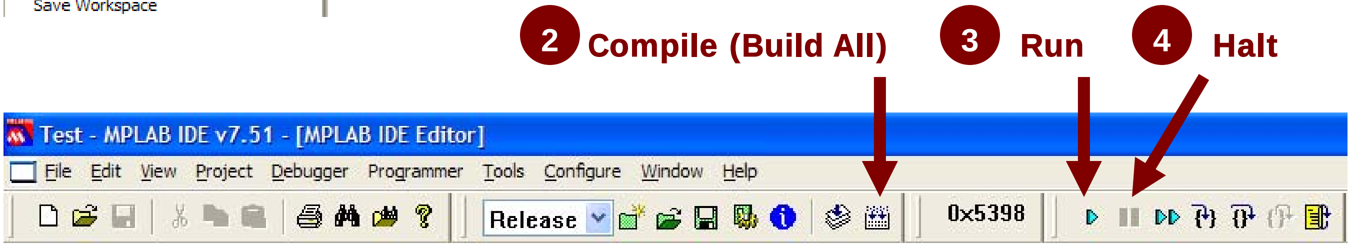
Procedure

On the class PC:
C:\RTC\101_ECP\Lab01\Lab01.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.


2 Compile (Build All)
3 Run
4 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button.

This will run the program in the simulator at full speed.

4 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

What just happened?

We took a pre-configured MPLAB®-IDE workspace, which included a complete project along with the configuration settings for the tools and window layout, and compiled the code contained in the project. After compiling the code, we ran it in the simulator that is built into MPLAB®. The simulator is capable of reproducing almost all of the functions of a PIC® microcontroller. The code itself doesn't do very much. We simply create and initialize a set of 6 variables. We then print out the size of these variables to the Sim Uart1 I/O window by using the printf() standard C library function. After stopping the code, we may then observe the contents of the variables and see how they are stored in the devices memory. For details on how to setup a project like this, please see Appendix A and Appendix B.

Results

Let’s take a look at what this code did when it was executed in the simulator. Some things might not make complete sense at this point, since they will be covered in detail later on in the class. Rest assured, we will cover all of this as we go forward.

Note that in the code below, I have removed some of the comments and changed the line spacing from the actual source code in order to save space.

Lab01.c

```c
01  /* Include the appropriate header (.h) file, depending on device used */
02  /* Example (for PIC24FJ128GA010): #include <P24FJ128GA010.h>          */
03  
04  #include <stdio.h>
05  
06  #define CONSTANT1 50
07  
08  // ********** VARIABLE DECLARATIONS **********
09  
10  char charVariable;
11  short shortVariable;
12  int intVariable;
13  long longVariable;
14  float floatVariable;
15  double doubleVariable;
16  
17  // ********** PROTOTYPE DEFINITIONS **********
18  
19  int main(void);
20  
21  /*******************************************************************************************
22  * Function Name:  void main(void)
23  *******************************************************************************************/
24  
25  int main (void)
26  {
27          charVariable = CONSTANT1;
28          intVariable = CONSTANT1;
29          longVariable = CONSTANT1;
30          floatVariable = CONSTANT1;
31          doubleVariable = CONSTANT1;
32  
33          printf("\nA character variable requires %d byte", sizeof( char ));
34          printf("\nA short variable requires %d bytes", sizeof( short ));
35          printf("\nAn integer variable requires %d bytes", sizeof( int ));
36          printf("\nA long variable requires %d bytes", sizeof( long ));
37          printf("\nA floating point variable requires %d bytes", sizeof( float ));
38          printf("\nA double variable requires %d bytes", sizeof( double ));
39  
40          while(1);
41  }
```

Line 4: Include header file for standard I/O routines. This header file is required because we will be using the printf() standard I/O library function later in this program.

Line 6: Define a constant called **`CONSTANT1`** whose value is 50. Constants will be discussed in the next section.

Lines 10-15: Here we declare six variables of different types. Note that the variables declared as **`short`** and **`long`** could have also been declared with their full type names of **`short int`** and **`long int`**, though very few C programmers write code that way.

Line 19: This is a function prototype for the **`main()`** function. Function prototypes will be discussed later.

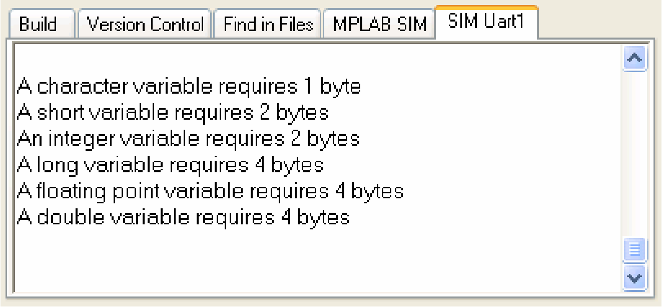
Line 25: This is the beginning of the **`main()`** function. Every C program must include this function. We will discuss the specifics of this later in the class. The application code will begin executing with the **`main()`** function, and that the syntax must look something like this:

```
int main(void)
{
    /* Main Application Code Here */
}
```

Lines 27-31: Here we initialize the variables we declared on lines 10-15. They are all initialized to have the same value, which is **`CONSTANT1`** or 50. (Remember that a **`char`** is just an 8-bit integer, so it is perfectly fine to assign an integer numeric value to it.)

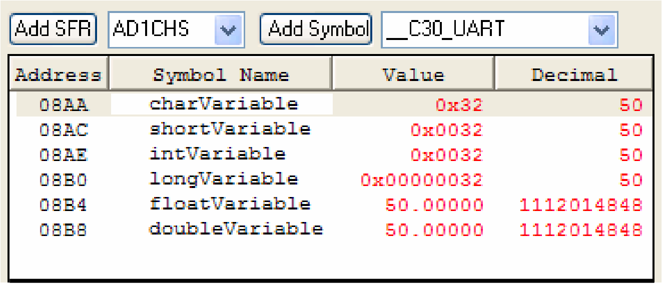
Lines 33-38: These lines print out the size of the data types to the Sim Uart1 window in MPLAB. The printf() function's syntax will be discussed later. It is not commonly used in an embedded environment, but it can be useful for debugging your code. In an actual microcontroller, printf() is often used to send a text string out the on-chip UART. We make use of the sizeof operator as a parameter to the printf() function, which will be discussed in the section on operators.

After the code completes execution, we will be able to observe the results in several ways:



```
A character variable requires 1 byte
A short variable requires 2 bytes
An integer variable requires 2 bytes
A long variable requires 4 bytes
A floating point variable requires 4 bytes
A double variable requires 4 bytes
```

5 The **SIM Uart1** window should show the text that is output by the program, indicating the sizes of C's data types in bytes.



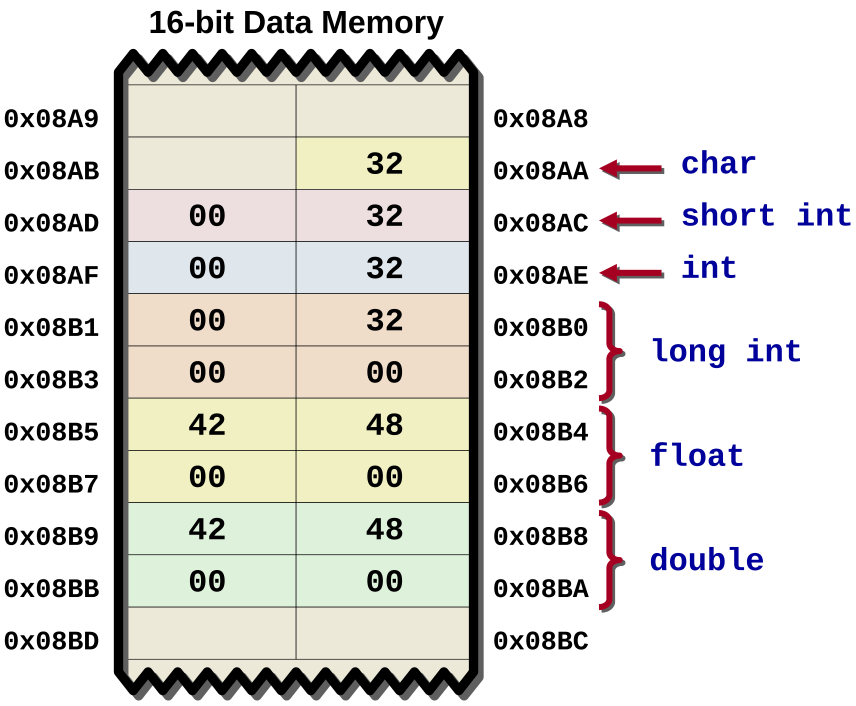
Address	Symbol Name	Value	Decimal
08AA	charVariable	0x32	50
08AC	shortVariable	0x0032	50
08AE	intVariable	0x0032	50
08B0	longVariable	0x00000032	50
08B4	floatVariable	50.00000	1112014848
08B8	doubleVariable	50.00000	1112014848

6 The watch window should show the values which are stored in the variables and make it easier to visualize how much space each one requires in data memory (RAM).

7 Variables in Memory

From the watch window above, we can see the addresses of the variables and we can see how many bytes they occupy by looking at the length of the data in the Value column. Based on this information, we can construct a memory map of our variables as shown on the next page.

Note that this example is laid out for a Microchip 16-bit microcontroller. For an 8-bit microcontroller, the data types will typically occupy the same number of bytes (some compilers define `int` to be only a single byte), but they would not be arranged side-by-side as shown.



16-bit Data Memory

0x08A9			0x08A8	
0x08AB		32	0x08AA	char
0x08AD	00	32	0x08AC	short int
0x08AF	00	32	0x08AE	int
0x08B1	00	32	0x08B0	long int
0x08B3	00	00	0x08B2	
0x08B5	42	48	0x08B4	float
0x08B7	00	00	0x08B6	
0x08B9	42	48	0x08B8	double
0x08BB	00	00	0x08BA	
0x08BD			0x08BC	

Multi-byte values stored in "Little Endian" format on PIC® microcontrollers

Conclusions

You have now seen how to declare variables:

Syntax

$type\ identifier_1,\ identifier_2,\ldots,identifier_n;$

You have also seen that different data types occupy different amounts of RAM. Since memory resources are relatively scarce in an embedded system, choosing the optimal data type for your variables is very important. This doesn't mean that you should always use the smallest type possible. Using a **char** in a 16-bit architecture might allow you to pack two 8-bit variables into a single RAM location, but it may also cause more code to be generated when manipulating those variables. As a general rule, the most highly optimized data type for a given architecture is the one that matches the data word width. For an 8-bit architecture, **char** is often the best. For a 16-bit architecture, an **int** is often best. Just keep in mind that not every compiler defines an **int** as 16-bits.

You have also seen how we can look at the contents of variables using the MPLAB®-IDE and that you can look at the value of the variable as the C compiler sees it, as well as the raw value as it is stored in data memory (RAM).

```c
/********************************************************************************
|  CLASS:         101_ECP - Getting Started with Embedded C Programming
|  PROGRAM:       Lab01.c
|  AUTHOR:        Denny Hopp
|  DATE:          13 DEC 2006
|  REQUIREMENTS:  (1) A heap is required for the printf() function
|                    (See handout for instructions on allocating a heap)
|  NOTES:         Code was written generically so that it may be used with any
|                 processor or compiler, though the MPLAB workspace has been
|                 configured to use MPLAB C30 with the PIC24FJ128GA010.
|
|  REVISION HISTORY:
|  01 MAY 2007    Rob Ostapiuk
|                 Updated to meet new lab coding standards
********************************************************************************/

/*-------------------------------------------------------------------------------
  HEADER FILES
-------------------------------------------------------------------------------*/
//#include <P24FJ128GA010.h>    // Processor specific header file
                                //   Not required here since we are not using
                                //   any processor specific features
#include <stdio.h>              // Standard I/O - required for printf() function

/*-------------------------------------------------------------------------------
  PROGRAM CONSTANTS
-------------------------------------------------------------------------------*/
#define CONSTANT1 50

/*-------------------------------------------------------------------------------
  VARIABLE DECLARATIONS
-------------------------------------------------------------------------------*/
char charVariable;
short shortVariable;            // Same as "short int"
int intVariable;
long longVariable;              // Same as "long int"
float floatVariable;
double doubleVariable;

/*-------------------------------------------------------------------------------
  FUNCTION PROTOTYPES
-------------------------------------------------------------------------------*/
void main(void);


/*===============================================================================
  FUNCTION:     main()
  DESCRIPTION:  Prints out the storage size of each variable
  PARAMETERS:   none
  RETURNS:      nothing
  REQUIREMENTS: none
===============================================================================*/
void main(void)
{
    /*---------------------------------------------------------------------------
      Initialize Variables
    ---------------------------------------------------------------------------*/
    charVariable = CONSTANT1;
    intVariable = CONSTANT1;
    longVariable = CONSTANT1;
    floatVariable = CONSTANT1;
    doubleVariable = CONSTANT1;

    /*---------------------------------------------------------------------------
      Print out storage size of each variable
    ---------------------------------------------------------------------------*/
    printf("A character variable requires %d byte\n", sizeof( char ));
    printf("A short variable requires %d bytes\n", sizeof( short ));
    printf("An integer variable requires %d bytes\n", sizeof( int ));
    printf("A long variable requires %d bytes\n", sizeof( long ));
    printf("A floating point variable requires %d bytes\n", sizeof( float ));
    printf("A double variable requires %d bytes\n", sizeof( double ));

    /*---------------------------------------------------------------------------
      Loop Forever
    ---------------------------------------------------------------------------*/
    while(1);
}
```

Lab 2

Symbolic Constants—Demonstration

Purpose

The purpose of this lab is to illustrate the difference between symbolic constants declared with the `const` keyword and those declared with the `#define` preprocessor directive.

Upon completion of this lab, you will see that constants declared with the `const` keyword will consume program memory locations, while constants declared with `#define` require no memory at all. You will also understand how both types of constants are displayed in watch windows within the MPLAB® Integrated Development Environment.

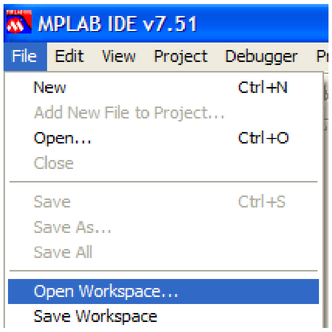
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab02\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab02\...

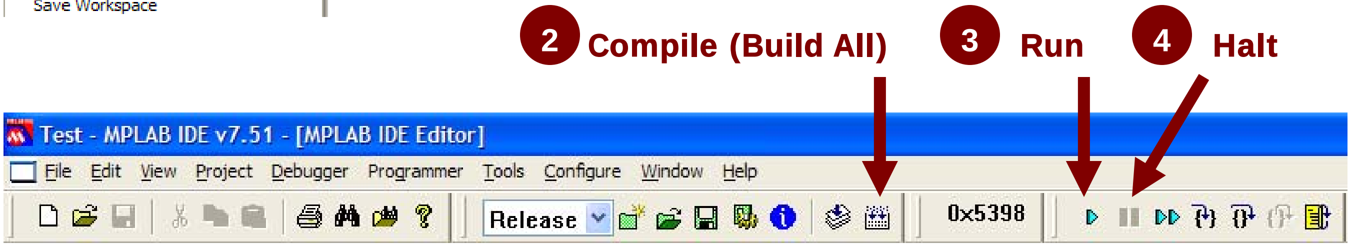
Procedure

On the class PC:
C:\RTC\101_ECP\Lab02\Lab02.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.


2 Compile (Build All)
3 Run
4 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button. This will run the program in the simulator at full speed.

4 Click on the **Halt** button. This will stop execution so that we may examine the variables and their values.

What just happened?

Just like the last lab, we took a pre-configured project and compiled its code. The program creates two constants. One constant is declared using the `const` keyword and the other is declared using the `#define` compiler directive. As described in the presentation, the constant declared with `const` will be placed in program memory as a "constant variable". The constant declared with `#define`, however requires no memory whatsoever, and we will see this in MPLAB once we simulate the code and observe the results in the watch window.

Results

The code really does very little and is described below.

Lab02.c

```c
01 //#include <p24fj128ga010.h>
02 #include <stdio.h>
03 
04 
05 #define CONSTANT1 0x33
06 const CONSTANT2 = 0xCC;
07 
08 
09 int variable1 = CONSTANT1;
10 int variable2;
11 //int variable2 = CONSTANT2;
12 // This cannot be done with a constant defined with "const"
13 
14 
15 int main(void)
16 {
17       variable2 = CONSTANT2;
18 
19 
20       printf("The first constant is 0x%X\n", CONSTANT1);
21       printf("The second constant is 0x%X\n", CONSTANT2);
22 
23 
24       while(1);
25 }
```

Line 5: A text substitution constant called `CONSTANT1` is created using the preprocessor directive `#define`.

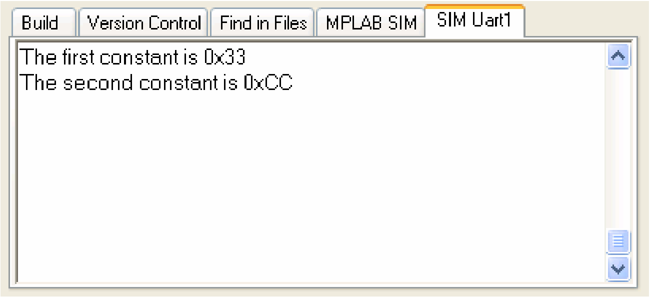
Line 6: A constant variable called `CONSTANT2` is declared using the `const` keyword.

Line 9: A variable called `variable1` is declared and initialized with the value of `CONSTANT1`.

Line 10: A variable called `variable2` is declared, but we cannot initialize it with `CONSTANT2` because a variable may not be initialized at declaration with another variable. `CONSTANT2` is a constant variable (a variable whose value may not be changed in code), and therefore it may not be used in this case.

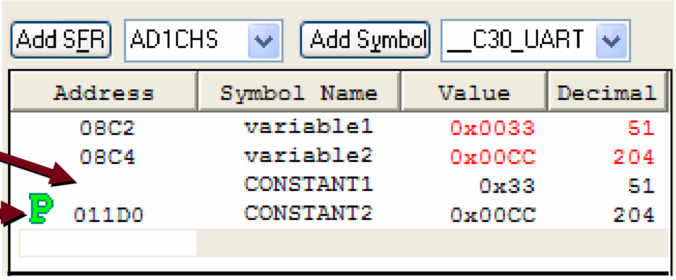
Line 17: Now that we are into executable code, we can now assign the value of the constant variable `CONSTANT1` to the variable `variable2`.

Lines 20-21: These lines simply print out the value of the two constants to the Sim UART1 IO window.



```
The first constant is 0x33
The second constant is 0xCC
```

5 The **SIM Uart1** window should show the text that is output by the program, indicating the values of the two symbolic constants in the code.

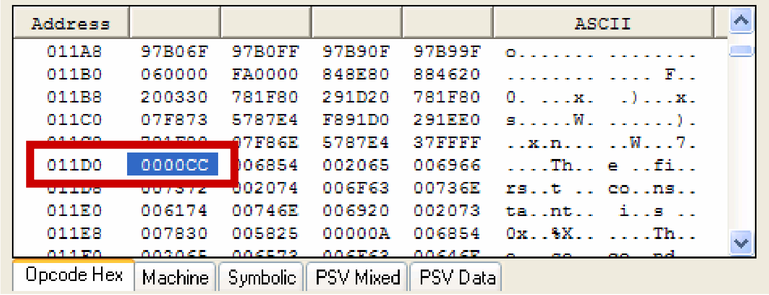


Address	Symbol Name	Value	Decimal
08C2	variable1	0x0033	51
08C4	variable2	0x00CC	204
	CONSTANT1	0x33	51
P 011D0	CONSTANT2	0x00CC	204

CONSTANT1 has no address

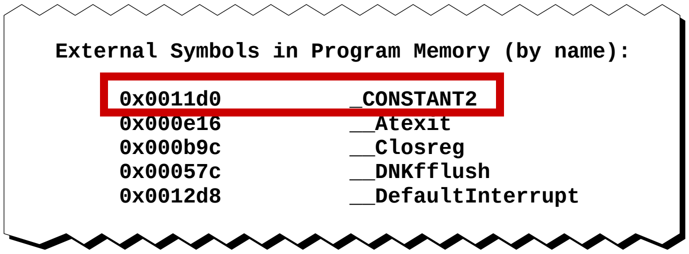
CONSTANT2 has a program memory address (P)

6 The watch window should show the two symbolic constants declared in code. **CONSTANT1** was declared with **#define**, and therefore uses no memory. **CONSTANT2** was declared with **const** and is stored as an immutable variable in flash program memory.



```
Address                                         ASCII
 011A8    97B06F  97B0FF  97B90F  97B99F  o....... ........
 011B0    060000  FA0000  848E80  884620  ........ .... F..
 011B8    200330  781F80  291D20  781F80  0. ...x.  .)...x.
 011C0    07F873  5787E4  F891D0  291EE0  s.....W. .......).
                  07F86E  5787E4  37FFFF  ..x.n... ..W...7.
 011D0    0000CC  006854  002065  006966  ....Th.. e ..fi..
                  002074  006F63  00736E  rs..t .. co..ns..
 011E0    006174  00746E  006920  002073  ta..nt..  i..s ..
 011E8    007830  005825  00000A  006854  0x..%X.. ....Th..
```

7 If we look in the program memory window, we can find **CONSTANT2** which was created with **const** at address 0x011D0 (as was shown in the watch window)



```
External Symbols in Program Memory (by name):

    0x0011d0                  _CONSTANT2
    0x000e16                  __Atexit
    0x000b9c                  __Closreg
    0x00057c                  __DNKfflush
    0x0012d8                  __DefaultInterrupt
```

CONSTANT1 does not appear anywhere in the map file!

8 If we open the map file (in the lab02 project directory), we can see that memory has been allocated for **CONSTANT2** at 0x011D0, but nothing has been allocated for **CONSTANT1**.

Conclusions

While there may be some circumstances when you need to declare a constant variable using `const`, in the overwhelming majority of cases, you will be better off using `#define`. In a microcontroller, with its limited memory resources, constants declared with `const` can quickly consume valuable program memory space, which could in some cases make the difference between using a part with 8K of memory or being forced into a more expensive part with 12K or 16K of memory. Constants declared with `#define` are handled by the preprocessor by substituting the text label with the constant's value before the code actually gets compiled. Therefore, no memory on the microcontroller is used to accommodate constants declared with `#define`. However, MPLAB® is still capable of displaying either kind of constant in a watch window if desired. Constants declared with `const` will be shown with their program memory address and a green 'P' next to it. Constants declared with `#define` will simply be shown without any address next to it.

Lab 3

`printf()` *Library Function—Demonstration*

Purpose

The purpose of this lab is to illustrate the use of the `printf()` standard C library function. While this function traditionally hasn't been used in embedded systems since its original purpose was to print text to the standard output device of a computer (monitor screen or printer), it has gained new popularity as many compilers have redirected it's output to the microcontroller's UART. When using the simulator, this function may be used to print text strings to a window within the MPLAB® Integrated Development Environment. This can be a powerful debugging technique as well as a convenient method for us to display the results of the programs we will be working on in this class.

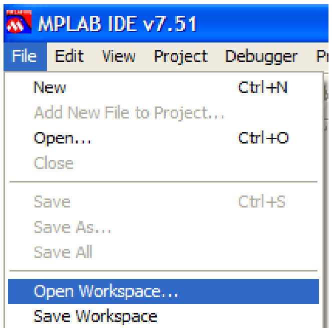
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab03\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab03\...

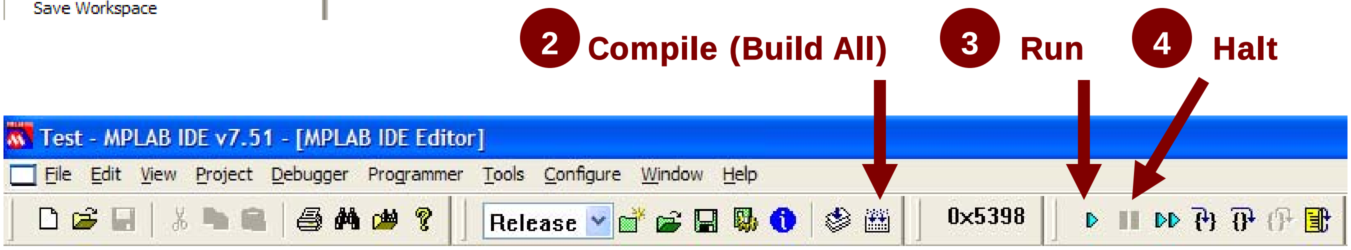
Procedure

On the class PC:
C:\RTC\101_ECP\Lab03\Lab03.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.


2 Compile (Build All)
3 Run
4 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button.

This will run the program in the simulator at full speed.

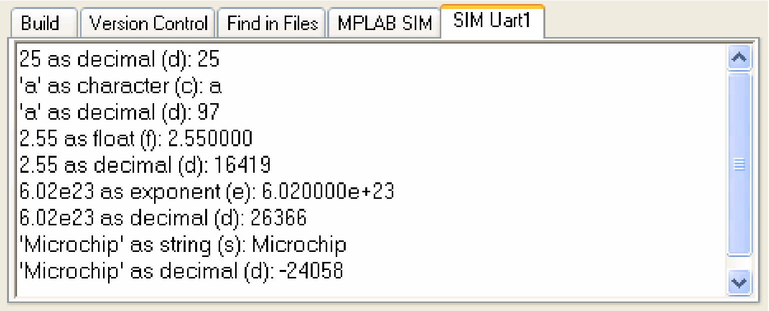
4 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

What just happened?

As before, we opened a preconfigured project, compiled the complete program and executed it. No code needed to be added or modified for this lab.

5 The SIM Uart1 output window will show the results of the program. The code used the `printf()` function to print out a variety of data types and data formats. We'll break down the steps on the next page.



```
25 as decimal (d): 25
'a' as character (c): a
'a' as decimal (d): 97
2.55 as float (f): 2.550000
2.55 as decimal (d): 16419
6.02e23 as exponent (e): 6.020000e+23
6.02e23 as decimal (d): 26366
'Microchip' as string (s): Microchip
'Microchip' as decimal (d): -24058
```

Results

Below is a table of the most commonly used control characters for the `printf()` function. Anywhere one of these characters is encountered after a '%' within a string, they will be replaced by the formatted value of its respective argument from the list following the string. The arguments may be literals or variables.

Conversion Character	Meaning
c	Single character
s	String (all characters until '\0')
d	Signed decimal integer
o	Unsigned octal integer
u	Unsigned decimal integer
x	Unsigned hexadecimal integer with lowercase digits (**1a5e**)
X	As **x**, but with uppercase digits (e.g. **1A5E**)
f	Signed decimal value (floating point)
e	Signed decimal with exponent (e.g. **1.26e-5**)
E	As **e**, but uses **E** for exponent (e.g. **1.26E-5**)
g	As **e** or **f**, but depends on size and precision of value
G	As **g**, but uses **E** for exponent

Syntax

```
printf(ControlString, arg1, arg2, … , argN)
```

Code Analysis

Below is an explanation of each **printf()** statement from the Lab3.c source file.

a. `printf("25 as decimal (d): %d\n", 25);`
This statement prints out the value 25 as a decimal integer (%d). The output correctly displays "25".

b. `printf("'a' as character (c): %c\n", 'a');`
This statement prints out the letter 'a' as a character (%c). The output correctly displays "a".

c. `printf("'a' as decimal (d): %d\n", 'a');`
This statement prints out the letter 'a' as a decimal integer (%d). Instead of displaying "a", the output displays the ASCII value of the character 'a', which is 97. This isn't necessarily incorrect as it will depend on what your code is trying to accomplish with the character or variable of type char.

d. `printf("2.55 as float (f): %f\n", 2.55);`
This statement prints out the value 2.55 as a floating point number (%f). The output correctly displays "2.550000". It is possible to specify the number of digits that should be displayed, but this was left as the default of 6 digits after the decimal point. (Default may vary among compilers.)

e. `printf("2.55 as decimal (d): %d\n", 2.55);`
This statement prints out the value 2.55 as a decimal number (%d). The output incorrectly displays “16419”. Because floating point values such as 2.55 are stored using the IEEE-754 format, it needs to be properly decoded to retrieve the correct value. In this case, we are displaying a portion of the raw IEEE-754 encoded value.

f. `printf("6.02e23 as exponent (e): %e\n", 6.02e23);`
This statement prints out the value 6.02 x 10^{23} as a signed real decimal value with an exponent (%e). The output correctly displays “6.020000e+23”.

g. `printf("6.02e23 as decimal (d): %d\n", 6.02e23);`
This statement prints out the vlaue 6.02 x 10^{23} as a decimal integer (%d). The output incorrectly displays “26366” which is part of the raw IEE-754 encoded floating point number.

h. `printf("'Microchip' as string (s): %s\n", "Microchip");`
This statement prints out the text “Microchip” as a string (%s). The output correctly displays “Microchip”.

i. `printf("'Microchip' as decimal (d): %d\n", "Microchip");`
This statement prints out the text “Microchip” as a decimal integer (%d). The output incorrectly displays the value -24058.

Conclusions

When using the **`printf()`** function, it is very important to pick the correct conversion character to use in the control string. If the wrong character is used, data will still be printed out, but it may not accurately represent the true value of the data in the argument list. Depending on the application, a simple mistake with the conversion character can have serious ocnsequences.

Lab 4

Operators—Hands-on Exercise

Purpose

The purpose of this lab is to illustrate the use of several arithmetic and logical operators of the C language.

In this exercise we will work with the various forms of the assignment operator, the basic arithmetic operators, the increment and decrement operators, and the bit operators. Upon completion of this exercise, you should understand how to code basic arithmetic and logical expression statements.

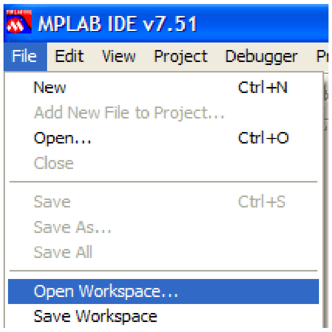
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab04\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab04\...

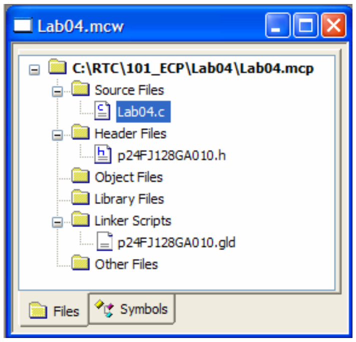
Procedure

On the class PC:
C:\RTC\101_ECP\Lab04\Lab04.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab04.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

Search the code for lines with the comment "`//### Your Code Here ###`". There will be additional comments above these lines with complete instructions, and in some cases there will be comments to the right with specific details regarding a particular line of code.
Note that comments with instructions for your tasks are surrounded by '#' to make them easy to spot.

Line 64

STEP 1: Add `charVariable1` to `charVariable2` and store the result in `charVariable1`. Algebraically speaking, this is equivalent to x = x + y. There are two ways of accomplishing this task in C. On **line 72**, perform this operation using the + operator (similar to the algebraic syntax). On **line 74**, perform this same operation again, but this time using the += operator.

Line 78

STEP 2: Increment `charVariable1`. There are several ways this could be done. Use the one that requires the least amount of typing. Algebraically, this is equivalent to x = x + 1.

Line 95

STEP 3: Use the conditional operator to set `longVariable1` equal to `intVariable1` if `charVariable1` is less than `charVariable2`. Otherwise, set `longVariable1` equal to `intVariable2`. If we were to do this the long way, it might look something like this:

```
if (charVariable1 < charVariable2)
    longVariable1 = intVariable1;
else
    longVariable1 = intVariable2;
```

However, here we want to make use of the conditional operator ‘ `? :` ‘ to perform the same task. Remember, the syntax of the conditional operator is:

Syntax

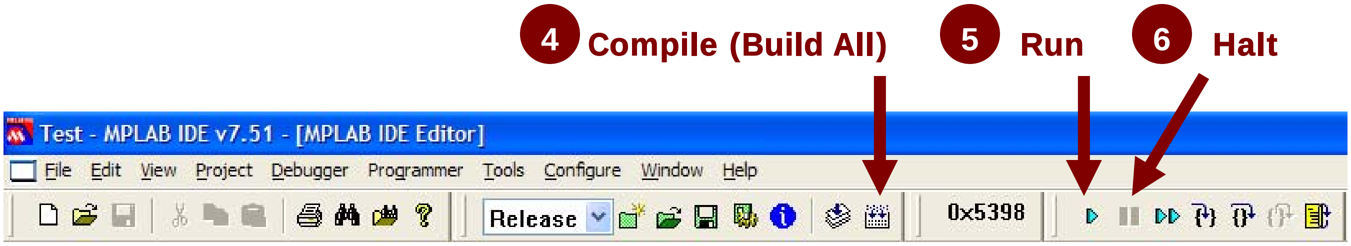
```
(test-expr) ? do-if-true : do-if-false;
```

Line 118

STEP 4: Shift `longVariable2` one bit to the right. There several ways this can be done, but the shortest is to use the appropriate compound assignment operator. Algebraically, this can be represented as: x = x / 2, but the shift operators in C will perform this operation much more efficiently than a divide operator could.

Line 129

STEP 5: Perform the logical operation (`longVariable2` *AND* `0x30`) and store the result back in longVariable2. Once again, the fastest way to do this is to use the appropriate compound assignment operator that performs the equivalent operation to: `longVariable2 = longVariable2 & 0x30`. If you need additional hints, take a look at the code below this step in the source file.


4 Compile (Build All)
5 Run
6 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

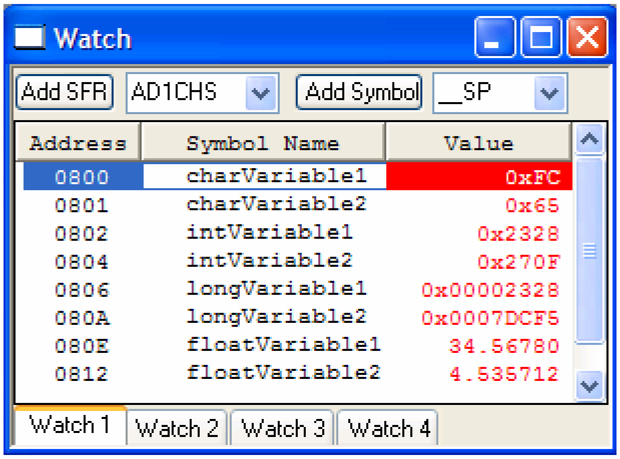
4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button. This will run the program in the simulator at full speed.

6 Click on the **Halt** button. This will stop execution so that we may examine the variables and their values.

Look at the variables in the watch window and take note of their values. Did the code change them the way you would expect?

Results



Address	Symbol Name	Value
0800	charVariable1	0xFC
0801	charVariable2	0x65
0802	intVariable1	0x2328
0804	intVariable2	0x270F
0806	longVariable1	0x00002328
080A	longVariable2	0x0007DCF5
080E	floatVariable1	34.56780
0812	floatVariable2	4.535712

The watch window will show the results you should have after running the entire program.

Now, let’s take a look at the code and see how this all came about.

Code Analysis

Up near the top, we declared several variables and gave them initial values:

```
char charVariable1 = 50;
char charVariable2 = 100;
int intVariable1 = 1000;
int intVariable2 = 10000;
long longVariable1 = 1000;
long longVariable2 = 2000;
float floatVariable1 = 34.5678;
float floatVariable2 = 156.78956;
```

Then, in the main() function, we perform a variety of arithmetic and logic operations on the variables. In step 1, you were asked to perform two addition operations where one of the variables was both an operand as well as the target for the result. The two lines should have looked something like this:

```
      charVariable1 = charVariable1 + charVariable2;
      charVariable1 += charVariable2;
```

The first one is the long way of doing this. It is perfectly correct to do it this way, but most C programmers prefer the second method, which uses the compound assignment operator ‘+=‘. Aside from the stereotype that C programmers like to write the most cryptic code possible, this method does require less typing, and also forces the compiler to recognize that `charVariable1` is both a source operand and the result destination of the operation. This can lead to more compact code with some compilers that don’t take the time to notice that the same variable is on both sides of the equals sign of the first method.

In step 2, you are asked to increment the variable `charVariable1`. There are at least three ways this can be done. The first two are similar to what we saw above:

```
      charVariable1 = charVariable1 + 1;
      charVariable1 += 1;
```

But the shortest way to do this is to use C’s increment operator ‘++’:

```
      charVariable1++;
```

or, in this case we could also use the prefix version:

```
      ++charVariable1;
```

Since no other operation is being carried out on the line, it doesn’t matter whether we use the prefix or postfix version of the increment operator. In all three cases, the result would be that `charVariable1`'s value would be incremented by one after the operation.

The next four lines of code illustrate the use of several other operators: subtraction (`-`), decrement (–), multiply compound assignment (`*=`) and divide compound assignment (`/=`). If you are having trouble understanding any of these, try stepping through the code to see how the variables change at each step.

In step 3, you are asked to use the conditional operator to set longVariable1 to intVariable1 if charVariable1 is less than charVariable2 and to intVariable2 otherwise. As shown in the procedure section, this could be written out the long way as:

```
if (charVariable1 < charVariable2)
    longVariable1 = intVariable1;
else
    longVariable1 = intVariable2;
```

Using the conditional operator, this could be coded as:

```
longVariable1 = (charVariable1 < charVariable2) ? intVariable1 : intVariable2;
```

The expression on the right of the equals sign will be equal to `intVariable1` if the expression before the question mark is true, and it will be equal to `intVariable2` if the expression is false.

The next line of code illustrates an example of the comma operator. Honestly, this isn't a particularly common way of using it. When we get to the section on `for` loops, you will see one of its more common uses.

In step 4, you are asked to shift `longVariable2` one bit to the right, which is the equivalent of dividing by two. As with other operations, there are several ways this could be accomplished:

```
      longVariable2 = longVariable2 >> 1;
      longVariable2 >>= 1;
```

The second method is the one we were going for here, but either one gets the job done.

Finally, in step 5, you were asked to perform a bitwise AND between longVariable2 and the literal value 0x30 and store the result in `longVariable2`. Once again, there are a few ways of doing this:

```
      longVariable2 = longVariable2 & 0x30
      longVariable2 &= 0x30
```

As before, the second solution is the one we were looking for. The remaining code consists of a few more examples of various operators.

Conclusions

Hopefully this gives you a feel for how the various C operators work. There was a strong emphasis on the compound assignment operators because that is the one area most new C programmers have the most difficulty with, but that experienced C programmers use most often.

The program itself doesn't do anything particularly useful. Its intent was to provide a platform to learn and experiment with C's operators without bogging you down with things not related to the topic at hand.

```
#include <stdio.h>              // Standard I/O - required for printf() function

/*-------------------------------------------------------------------------------
  VARIABLE DECLARATIONS
-------------------------------------------------------------------------------*/
char charVariable1 = 50;
char charVariable2 = 100;
int intVariable1 = 1000;
int intVariable2 = 10000;
long longVariable1 = 1000;
long longVariable2 = 2000;
float floatVariable1 = 34.5678;
float floatVariable2 = 156.78956;
/*-------------------------------------------------------------------------------
  FUNCTION PROTOTYPES
-------------------------------------------------------------------------------*/
void main(void);

/*===============================================================================
  FUNCTION:     main()
===============================================================================*/
void main(void)
{
        /*---------------------------------------------------------------------------
          Standard Mathematical Operators
        ---------------------------------------------------------------------------*/
        charVariable1 = charVariable1 + charVariable2;
        charVariable1 += charVariable2;
        charVariable1++;
        intVariable1 = intVariable2 - intVariable1;
        intVariable2--;
        longVariable2 *= longVariable1;
        floatVariable2 /= floatVariable1;
        /*---------------------------------------------------------------------------
          Conditional Operator
        ---------------------------------------------------------------------------*/
        longVariable1 = (charVariable1 < charVariable2) ? intVariable1 : intVariable2;
        /*---------------------------------------------------------------------------
          Comma Operator
        ---------------------------------------------------------------------------*/
        longVariable2 = (charVariable1++ , charVariable2++);
        /*---------------------------------------------------------------------------
          Bit Shift Operator
        ---------------------------------------------------------------------------*/
        longVariable2 >>= 1;        //Shift longVariable2 one bit to the right
        /*---------------------------------------------------------------------------
          Logical AND Operators
        ---------------------------------------------------------------------------*/
        longVariable2 &= 0x30;        //longVariable2 = longVariable2 & 0x30
        /*---------------------------------------------------------------------------
          Logical Inclusive OR Operators (both lines perform the same operation)
        ---------------------------------------------------------------------------*/
        //longVariable2 = longVariable2 | 0x0F;
        longVariable2 |= 0x0F;
        /*---------------------------------------------------------------------------
          Logical Exclusive OR Operators (both lines perform the same operation)
        ---------------------------------------------------------------------------*/
        //longVariable2 = longVariable2 ^ 0x03;
        longVariable2 ^= 0x03;
        /*---------------------------------------------------------------------------
          Loop Forever and repeatedly increment longVariable2
        ---------------------------------------------------------------------------*/
        while(1)
        {
                longVariable2++;
        }
}
```

Lab 5

Making Decisions (if)—Hands-on Exercise

Purpose

The purpose of this lab is to illustrate the use of the if statement to make decisions in code. An if statement will execute a block of code if some specified condition is met. The goal of this lab is for you to become comfortable with the if statement syntax and how you create the condition expressions.

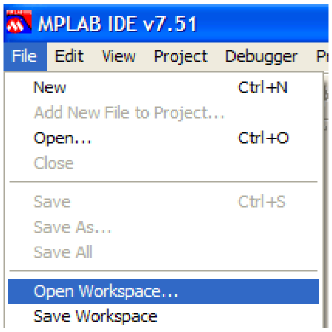
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab05\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab05\...

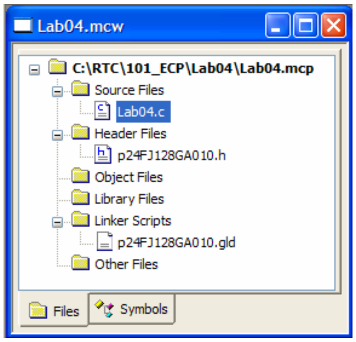
Procedure

On the class PC:
C:\RTC\101_ECP\Lab05\Lab05.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab05.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

Search the code for lines with the comment "`//### Your Code Here ###`". There will be additional comments above these lines with complete instructions, and in some cases there will be comments to the right with specific details regarding a particular line of code.
Note that comments with instructions for your tasks are surrounded by '`#`' to make them easy to spot.

Line 64

STEP 1: Increment intVariable1 if BOTH the following conditions are true:

- floatVariable2 is greater than or equal to floatVariabe1
- charVariable2 is greater than or equal to charVariable1

Remember to use parentheses to group operations. This step will require you to use logical operators such as '&' and relational operators such as '>='.

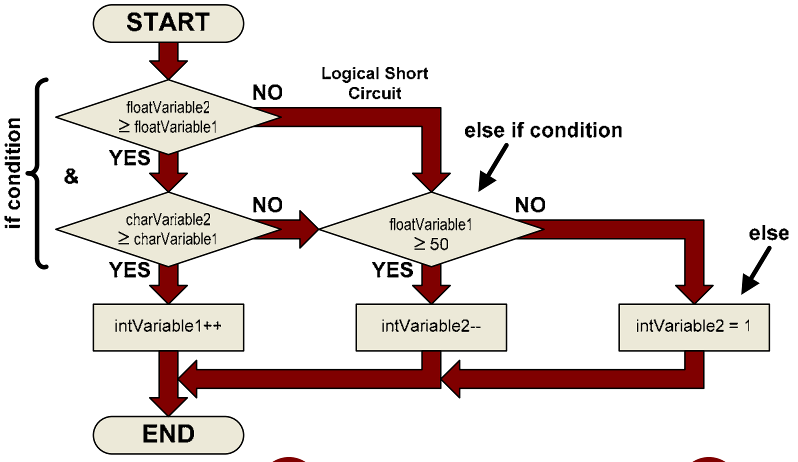
Line 77

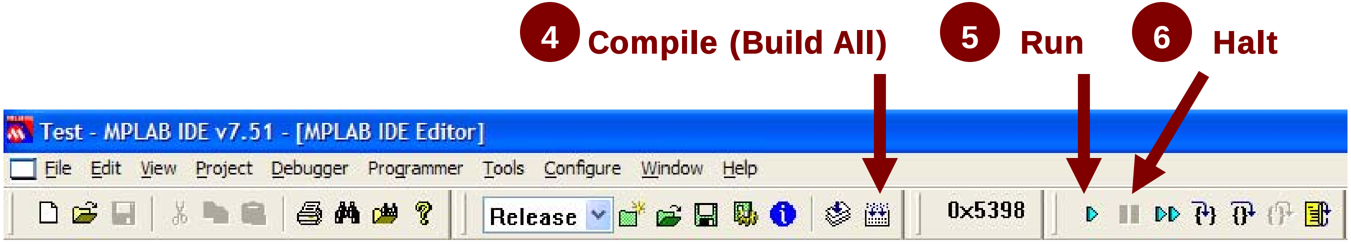
STEP 2: If the above is not true and floatVariable1 is greater than 50 then decrement intVariable1 (Hint: else if).

Line 88

STEP 3: If neither of the above is true, set charVariable2 equal to 1. (Hint: else)

This code you are writing implements the flowchart below:




4 Compile (Build All)

5 Run

6 Halt

4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

This will run the program in the simulator at full speed.

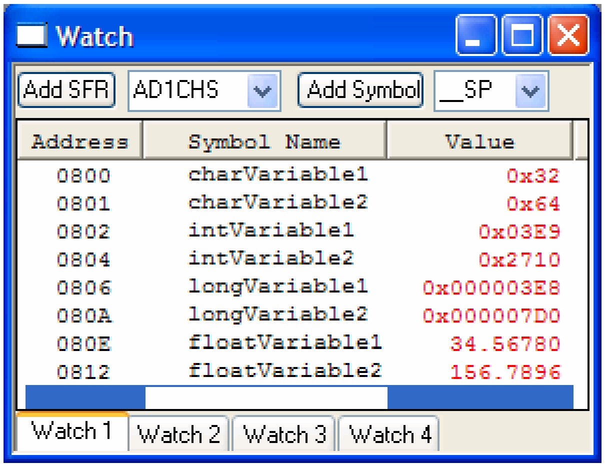
6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After you build the code, run it and then stop it, you should see the results at right in your watch window.

Next, we will cover the code step by step.



Address	Symbol Name	Value
0800	charVariable1	0x32
0801	charVariable2	0x64
0802	intVariable1	0x03E9
0804	intVariable2	0x2710
0806	longVariable1	0x000003E8
080A	longVariable2	0x000007D0
080E	floatVariable1	34.56780
0812	floatVariable2	156.7896

Code Analysis

(NOTE: Line numbers correspond to those in the provided solution file.)

Lines 35-42:

As in previous labs, we have created several variables that we will manipulate to illustrate the main ideal of the lab.

```
char charVariable1 = 50;
char charVariable2 = 100;
int intVariable1 = 1000;
int intVariable2 = 10000;
long longVariable1 = 1000;
long longVariable2 = 2000;
float floatVariable1 = 34.5678;
float floatVariable2 = 156.78956;
```

Lines 70-75:

STEP 1 instructed you to write code that will increment `intVariable1` if both of two specified conditions are true. This can easily be implemented as an if statement, with two conditions connected by the logical AND operator.

```
if((floatVariable2 >= floatVariable1) && (charVariable2 >= charVariable1))
{
      intVariable1++;
}
```

In order for `intVariable1` to be incremented, both of the conditions must be true: `floatVariable2` must be greater than or equal to `floatVariable1`, and `charVariable2` must be greater than or equal to `charVariable1`. Note that the operator connecting the two conditions is the logical AND ‘&&’. Make sure you use the double ampersand, otherwise it will not function properly in all conditions.

Lines 82-87:

STEP 2 instructed you to decrement `intVariable2` if the previous condition (STEP 1) was not true. This can most easily be accomplished by using an `else if` statement connected to the `if` statement above.

```
else if(floatVariable1 > 50)
{
      intVariable2--;
}
```

Since this `else if` statement follows the `if` statement above, it will check its condition only if the one above is false. So, for `intVariable2` to get decremented the following conditions must be met: `floatVariable2` must be less than `floatVariable1`, and `charVariable2` must be less than `charVariable1`, and `floatVariable1` must be greater than 50.

Lines 95-99:

STEP 2 instructed you to set `charVariable2` equal to 1 if all the other conditions were false. This can be done by using an `else` statement after the `if` and `else if` statements above.

```
else
{
      charVariable2 = 1;
}
```

At this point, `charVariable2` will only be set equal to 1 if `floatVariable2` is less than `floatVariable1` and `charVariable2` is less than `charVariable1` and `floatVariable1` is less than or equal to 50.

Conclusions

The if statement makes it possible to execute blocks of code only when some specified set of conditions are met. When used in conjunction with the else if and else statements (which can only be used following an if statement), it is possible to drill down through several levels of conditions and execute a different block of code for each level.

Complete Lab5 minimal source code without extra comments or unused elements:

```
#include <stdio.h>              // Standard I/O - required for printf() function

/*-------------------------------------------------------------------------------
  VARIABLE DECLARATIONS
-------------------------------------------------------------------------------*/
char charVariable1 = 50;
char charVariable2 = 100;
int intVariable1 = 1000;
int intVariable2 = 10000;
long longVariable1 = 1000;
long longVariable2 = 2000;
float floatVariable1 = 34.5678;
float floatVariable2 = 156.78956;

/*-------------------------------------------------------------------------------
  FUNCTION PROTOTYPES
-------------------------------------------------------------------------------*/
void main(void);

/*===============================================================================
  FUNCTION:    main()
===============================================================================*/
void main(void)
{
      if((floatVariable2 >= floatVariable1) && (charVariable2 >= charVariable1))
      {
            intVariable1++;
      }
      else if(floatVariable1 > 50)
      {
            intVariable2--;
      }
      else
      {
            charVariable2 = 1;
      }
      while(1);
}
```

Lab 6

Making Decisions (switch)—Hands-on Exercise

Purpose

The purpose of this lab is to illustrate the use of the switch statement to make decisions in code. A switch statement will execute one (or more) blocks of code depending on which condition is met. The goal of this lab is for you to become comfortable with the switch statement syntax and how you create the condition expressions.

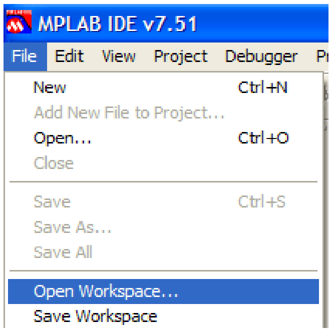
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab06\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab06\...

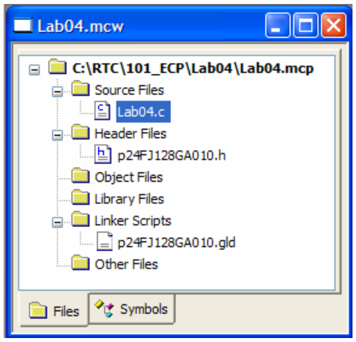
Procedure

On the class PC:
C:\RTC\101_ECP\Lab06\Lab06.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab06.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

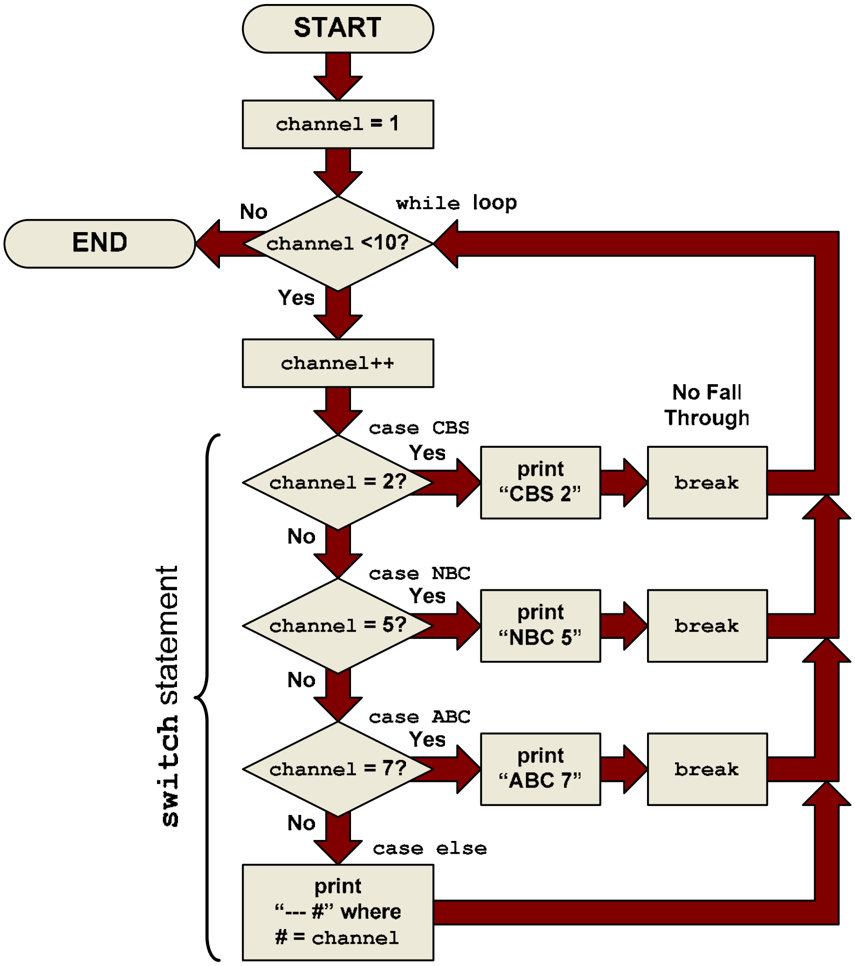
3 Edit source code as instructed in the comments

The task for this lab is to create a switch statement that will print out a particular string depending on the value of the control variable. Like some of the examples in the presentation, we will use Chicago TV channels and their American network affiliations as our data. (Feel free to localize the code to print out "CBC", "BBC", "Telemundo" or whatever you like.) Note that some constants have been defined to equate the network's initials with the TV channel number (CBS = 2, NBC = 5, ABC = 7).

The main loop of the program will increment the channel variable from 1 to 10. Our task is to print out either the network initials with its associated channel, or three dashes followed by the channel if there is no network affiliation. (See flow chart on next page).

STEP 1:
Open a `switch` statement on the variable `channel`. `channel` is our control variable, which will be incremented from 1 to 10 in the main loop. During each pass, we will use the `switch` statement to print out the appropriate string based on the value of `channel`.

STEP 2:
Write case for `channel` = CBS (CBS is a constant defined to equal 2). There are two things that need to be done here. First, we need to start a case block, and then within the block we need to print out the string "CBS 2". There are two ways to do this. One would be to simply do `print("CBS 2\n")`. While this will work in



this circumstance, while the constant CBS is defined to be 2, what would happen if we changed the constant at the top of the file to be 9? We would correctly get to this point when channel = 9, but we would incorrectly print out "CBS 2". So the better way to code this is to use the channel variable in our print statement. Remember the syntax for printf:

Syntax

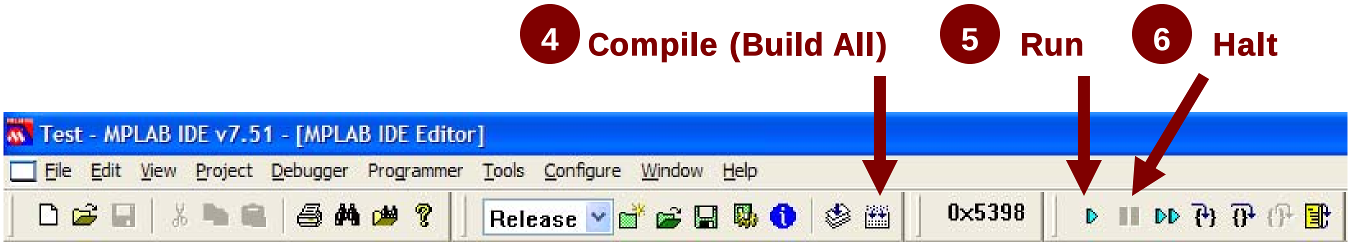
```
printf(ControlString, arg1, arg2, … , argN)
```

You can use %d as the placeholder in your string for the `channel` variable which would be the only argument used.

STEP 3:
Write the case for `channel` = NBC (NBC is a constant defined to equal 5). This step should look identical to step 2, but with the appropriate values used for NBC.

STEP 4:
Write the case for `channel` = ABC (ABC is a constant defined to equal 7). This step should look identical to step 2, but with the appropriate values used for ABC.

STEP 5:
Write the default case. If `channel` is anything other than those listed above, this is what should be done. For these cases, you need to print the string “—- #” where # is the `channel` number (value of the channel variable). For example, if `channel` = 6, you should print “—- 6”.


4 Compile (Build All)
5 Run
6 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

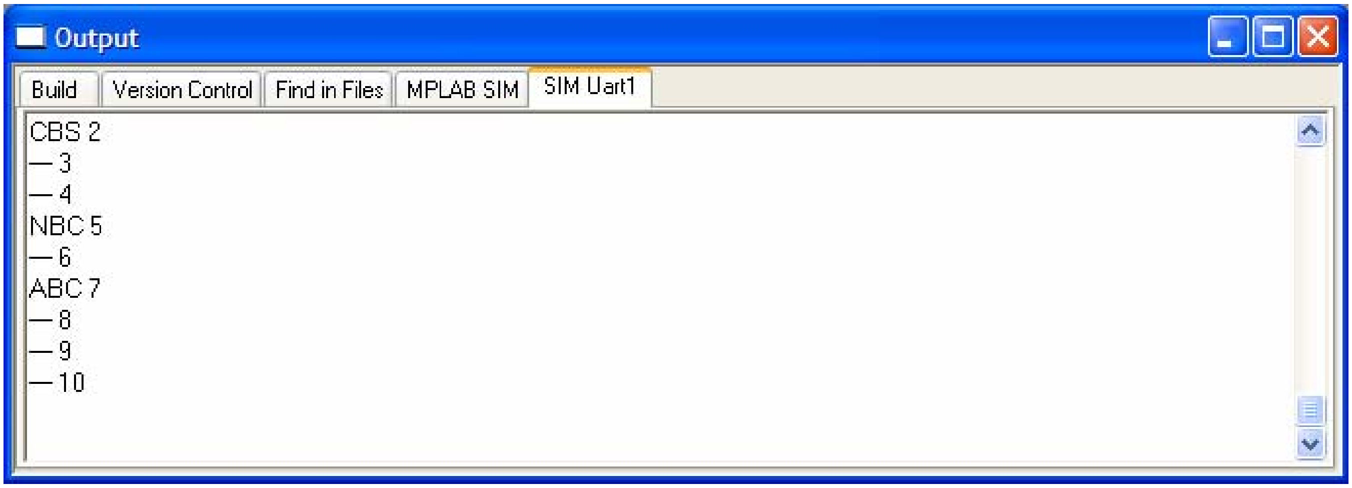
This will run the program in the simulator at full speed.

6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After you build the code, run it and then stop it, you should see the following text in the Sim Uart1 I/O output window:



```
CBS 2
— 3
— 4
NBC 5
— 6
ABC 7
— 8
— 9
— 10
```

Code Analysis

(NOTE: Line numbers correspond to those in the provided solution file.)

Lines 78-80:
STEP 1: The instructions asked you to open up a switch statement on the variable `channel`. Basically, all you need to do is write the first line(s) of code that are required to start off a switch statement with channel as the

control variable:

```
          switch(channel)
          {
```

This is just following the syntax definition for a switch statement. The variable channel is the one that will be evaluated for each of the case blocks that follow below.

Lines 85-91:
STEP 2 instructed you to write the case for channel = CBS (using the predefined constants from above). The first part of this step is to open up the case block with the case keyword and the value that is to be matched with the channel variable. Next, within the block, we need to print out “CBS 2” using the printf function, and finally finish up the block with a break statement to prevent fall through to the next case.

```
      case CBS:
      {
            printf("CBS %d\n", channel);
            break;
      }
```

The first line makes it so that this block will only be executed when channel = CBS. We then use a compound statement (enclosed by curly braces) so that we can have multiple instructions as part of this case block. The printf function makes use of the channel variable so that if we change the value of the constant CBS above, this will correctly print out the new channel number associated with this network. Finally we use a break statement to force us out of the switch block since we have already handled the current value of channel. If we eliminated the break statement, we would fall through to the next case and execute it, and so on until we hit the end of the switch block.

Lines 98-104:
STEP 3 will look just like step 2, but here we must substitute the appropriate text and values for NBC.

```
      case NBC:
      {
            printf("NBC %d\n", channel);
            break;
      }
```

Lines 111-117:
STEP 4 will look just like step 2, but here we must substitute the appropriate text and values for ABC.

```
      case NBC:
      {
            printf("NBC %d\n", channel);
            break;
      }
```

Lines 126-130:
STEP 5 completes the switch statement with the default case. If none of the above cases are true, this is the one that will get executed. Here, we simply want to print out “—-#” where # is the current channel number. Unlike the cases above, we don’t need a break statement here since there is nowhere left to fall through—we are already at the end.

```
      default:
      {
            printf("--- %d\n", channel);
      }
```

Conclusions

The switch statement provides a more elegant way to conditionally execute blocks of code based on multiple criteria than the if statement. The only drawback is that the case conditions must be constants, or some value that may be evaluated at compile time, whereas the if statement allows variables to be used as part of its conditions.

Complete Lab5 minimal source code without extra comments or unused elements:

```
#include <stdio.h>              // Standard I/O - required for printf() function

/*-------------------------------------------------------------------------------
  PROGRAM CONSTANTS
-------------------------------------------------------------------------------*/
#define CBS 2
#define NBC 5
#define ABC 7

/*-------------------------------------------------------------------------------
  VARIABLE DECLARATIONS
-------------------------------------------------------------------------------*/
int channel = 1;

/*-------------------------------------------------------------------------------
  FUNCTION PROTOTYPES
-------------------------------------------------------------------------------*/
void main(void);

/*===============================================================================
  FUNCTION:     main()
===============================================================================*/
void main(void)
{
      while(channel < 10)
      {
            channel++;
            switch(channel)
            {
                  case CBS:
                  {
                        printf("CBS %d\n", channel);
                        break;
                  }
                  … //next two cases look similar to the one for CBS
                  default:
                  {
                        printf("--- %d\n", channel);
                  }
            }
      }
      while(1);
}
```

Lab 7

Loops—Hands-on Exercise

Purpose

The purpose of this lab is to illustrate the use of the various looping mechanisms available to us in C. We will work with the for loop, while loop and do...while loop in this exercise. When you complete this lab, you should be able to create iterative code that will test its entry and/or exit conditions at the appropriate point in the algorithm.

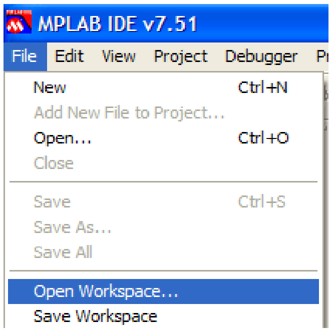
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab07\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab07\...

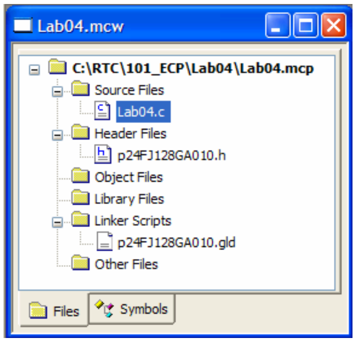
Procedure

On the class PC:
C:\RTC\101_ECP\Lab07\Lab07.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab07.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

In this lab, we have examples of the three loops, with an extra example of the do...while loop to illustrate the situation where only one pass will occur. To understand the program flow, see the flowchart on the following page.

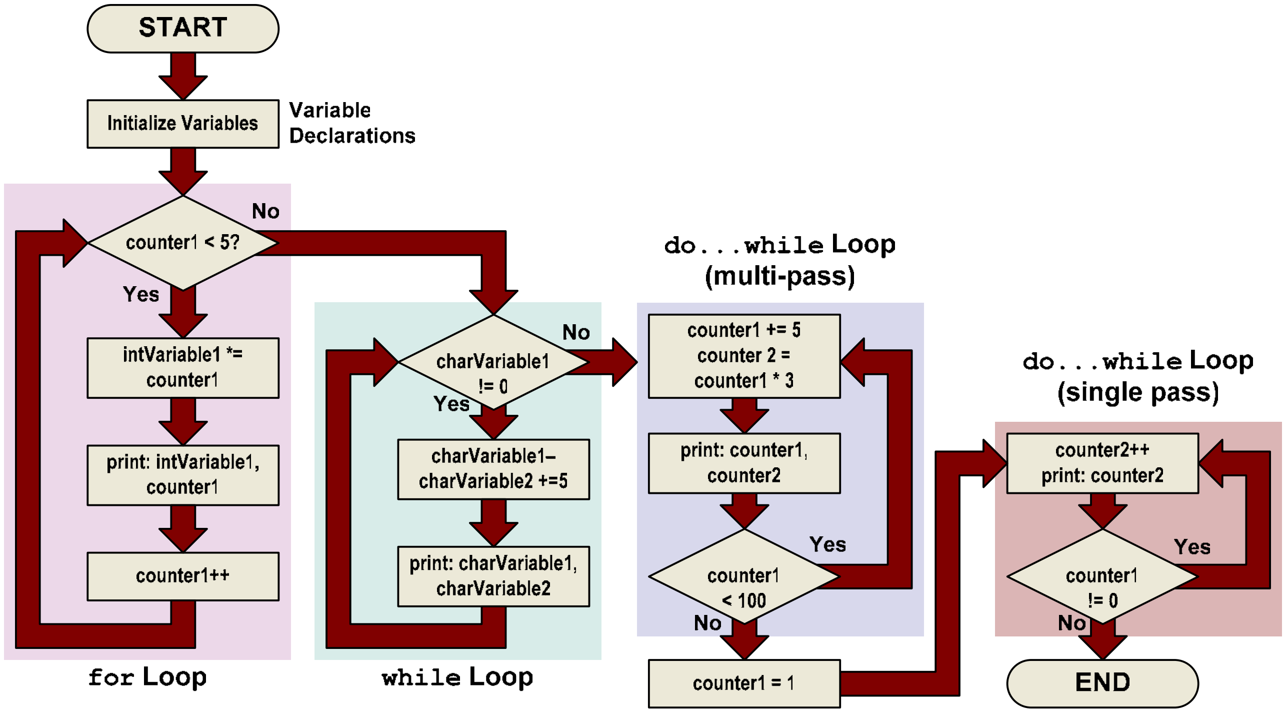
STEP 1:
Create a for loop to iterate through the block of code below. The loop should do the following:

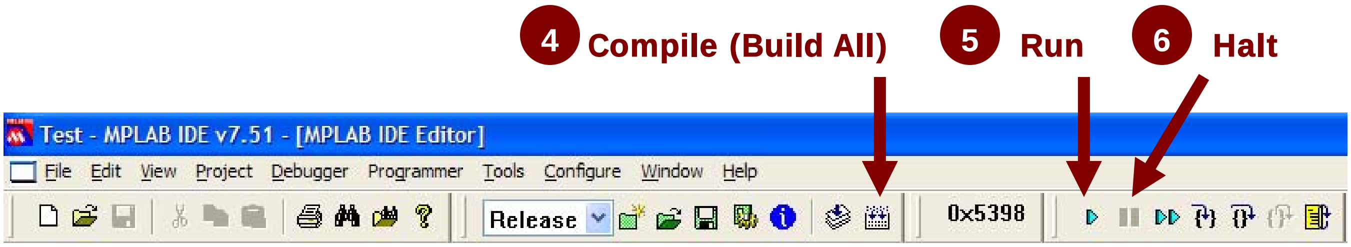
- Initialize counter1 to 1
- Loop as long as counter1 is less than 5
- Increment counter1 on each pass of the loop

(HINT: for (init, test, action) { … }

STEP 2:
Create a while loop to iterate through the block of code below. The loop should run until charVariable1 is 0.
(HINT: while(condition)) { … }

STEP 3:
Create a do...while loop to iterate through the block of code below. The loop should run until counter1 is greater than 100.
(HINT: do { … } while(condition);


START
Initialize Variables
Variable Declarations
counter1 < 5?
No
Yes
intVariable1 *= counter1
print: intVariable1, counter1
counter1++
for Loop
charVariable1 != 0
No
Yes
charVariable1– charVariable2 +=5
print: charVariable1, charVariable2
while Loop
do...while Loop (multi-pass)
counter1 += 5 counter 2 = counter1 * 3
print: counter1, counter2
counter1 < 100
Yes
No
counter1 = 1
do...while Loop (single pass)
counter2++ print: counter2
counter1 != 0
Yes
No
END


4 Compile (Build All)
5 Run
6 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

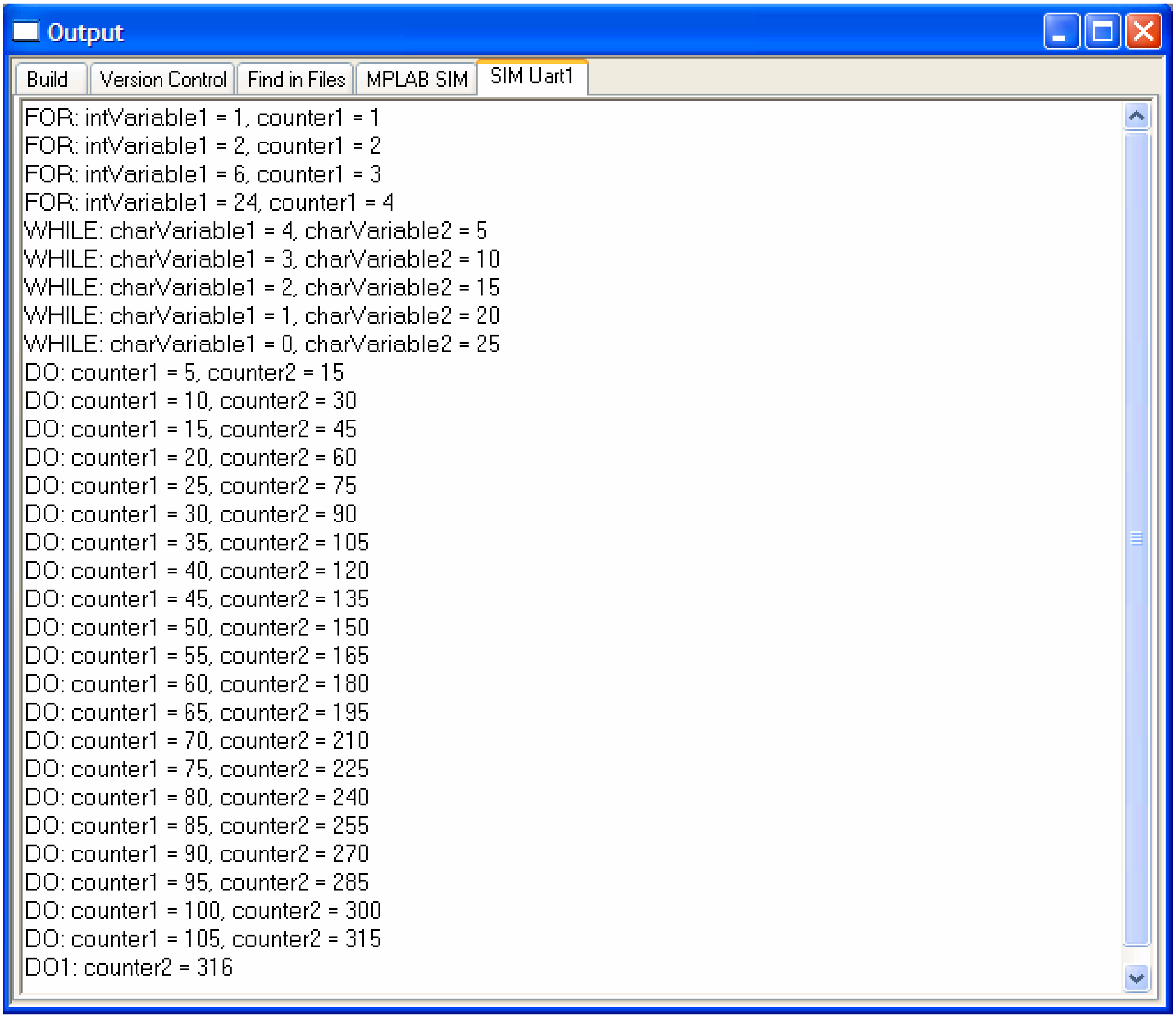
This will run the program in the simulator at full speed.

6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After building, executing and stopping your code, the Sim Uart1 I/O window should resemble the screenshot on the next page, where the relevant variables are printed out in each iteration of the respective loop. Note that in the very last do...while loop (you didn't write code for this one), it was setup such that the condition was not met when the loop started, but since the condition check doesn't occur until the end of a loop iteration, the loop did execute one time. This is why the last line in the output window says "DO1: counter2 = 316".



```
FOR: intVariable1 = 1, counter1 = 1
FOR: intVariable1 = 2, counter1 = 2
FOR: intVariable1 = 6, counter1 = 3
FOR: intVariable1 = 24, counter1 = 4
WHILE: charVariable1 = 4, charVariable2 = 5
WHILE: charVariable1 = 3, charVariable2 = 10
WHILE: charVariable1 = 2, charVariable2 = 15
WHILE: charVariable1 = 1, charVariable2 = 20
WHILE: charVariable1 = 0, charVariable2 = 25
DO: counter1 = 5, counter2 = 15
DO: counter1 = 10, counter2 = 30
DO: counter1 = 15, counter2 = 45
DO: counter1 = 20, counter2 = 60
DO: counter1 = 25, counter2 = 75
DO: counter1 = 30, counter2 = 90
DO: counter1 = 35, counter2 = 105
DO: counter1 = 40, counter2 = 120
DO: counter1 = 45, counter2 = 135
DO: counter1 = 50, counter2 = 150
DO: counter1 = 55, counter2 = 165
DO: counter1 = 60, counter2 = 180
DO: counter1 = 65, counter2 = 195
DO: counter1 = 70, counter2 = 210
DO: counter1 = 75, counter2 = 225
DO: counter1 = 80, counter2 = 240
DO: counter1 = 85, counter2 = 255
DO: counter1 = 90, counter2 = 270
DO: counter1 = 95, counter2 = 285
DO: counter1 = 100, counter2 = 300
DO: counter1 = 105, counter2 = 315
DO1: counter2 = 316
```

Code Analysis

(NOTE: Line numbers correspond to those in the provided solution file.)

Lines 73-79:
STEP 1 There are three parts to the creation of a for loop:

```
    for( counter1 = 1 ; counter1 < 5 ; counter1++)
    {
         intVariable1 *= counter1;
         printf("FOR: intVariable1 = %d, counter1 = %d\n", intVariable1, counter1);
    }
```

The first part is to initialize the loop count variable. This is done as the first for loop parameter statement. Next, we need to specify the condition under which the loop will continue looping. This is the second parameter. Remember, the loop will continue as long as this statement is true., and it is tested at the top of each loop iteration. Finally, we need to specify some action to take for each iteration of the loop. This is the third parameter where we increment the variable counter1. So, the code between the curly braces will repeat until counter1 >= 5. During each iteration of the loop, the variable counter1 will be incremented.

Lines 89-96:
STEP 2 had you create a while loop. This is a loop where the condition is tested at the top of the loop, so if the condition is not met, the loop will never execute. A while loop is similar to a for loop, and in fact it is a special case of a for loop (equivalent to for (; condition;) {…}). The only thing you specify in a while loop is the exit condition. Any loop counting will have to be conducted manually within the loop itself.

```
    while( charVariable1 != 0)
    {
         charVariable1--;
         charVariable2 += 5;
         printf("WHILE: charVariable1 = %d, charVariable2 = %d\n", charVariable1,
                        charVariable2);
    }
```

Like the for loop, the while loop will continue to execute as long as the condition is true. Since we want to execute this code until charVariable1 is 0, we simply specify a condition of charVariable1 not equal to 0.

Lines 107-115:
STEP 3 had you create a do...while loop. This is similar in concept to the while loop, in that the only thing we specify is the condition. However, the do loop checks its condition at the end of a loop iteration. Therefore, it is possible to execute the loop once, even if the condition is false.

```
    do
    {
        counter1 += 5;
        counter2 = counter1 * 3;
        printf("DO: counter1 = %d, counter2 = %d\n", counter1, counter2);
    } while(counter1 <= 100);
```

The do loop starts with the do keyword. Since we are executing multiple statements in the loop, we must enclose them in curly braces. Finally, the do loop ends with the keyword while followed by the loop condition. In this case, the code will continue looping until counter1 is greater than 100. Depending on how we increment counter1, it is possible for it to be greater than 100 on its final iteration.

Conclusions

C provides tremendous flexibility when it comes to loops. The for loop makes it possible to perform loop overhead tasks outside of the main body of the code block, by allowing variable initialization and modification within its parameter list. The while loop and do...while loop are just as flexible, but require more code in the body of the loop to perform the same actions. The while loop (and the for loop) checks its condition at the top of the loop, so if the condition isn't true at the beginning, it will never execute. In contrast, the do...while loop checks its condition at the end of the loop, so you will always have at least one loop iteration, even if the loop condition is not true from the start.

```
#include <stdio.h>              // Standard I/O - required for printf() function

/*-------------------------------------------------------------------------------
  PROGRAM CONSTANTS
-------------------------------------------------------------------------------*/
#define CONSTANT1 50

/*-------------------------------------------------------------------------------
  VARIABLE DECLARATIONS
-------------------------------------------------------------------------------*/
char charVariable1 = 5;
char charVariable2 = 0;
int intVariable1 = 1;
int intVariable2 = 10000;
long longVariable1 = 1000;
long longVariable2 = 2000;
float floatVariable1 = 34.5678;
float floatVariable2 = 156.78956;
int counter1;
int counter2;

/*-------------------------------------------------------------------------------
  FUNCTION PROTOTYPES
-------------------------------------------------------------------------------*/
void main(void);

/*===============================================================================
  FUNCTION:     main()
===============================================================================*/
void main(void)
{
    for( counter1 = 1 ; counter1 < 5 ; counter1++)
    {
         intVariable1 *= counter1;
         printf("FOR: intVariable1 = %d, counter1 = %d\n", intVariable1, counter1);
    }  //end for

    while( charVariable1 != 0)
    {
         charVariable1--;
         charVariable2 += 5;
         printf("WHILE: charVariable1 = %d, charVariable2 = %d\n", charVariable1, charVariable2);
    }  //end while

    counter1 = counter2 = 0;        //Clear variables used in for loop earlier
    do
    {
        counter1 += 5;
        counter2 = counter1 * 3;
        printf("DO: counter1 = %d, counter2 = %d\n", counter1, counter2);
    } while(counter1 <= 100);
    //end do...while

    counter1 = 0;                  //Clear variable used in previous loop
    do
    {
        counter2++;
        printf("DO1: counter2 = %d\n", counter2);
    } while(counter1 != 0);
    //end do...while
}
```

Lab 8

Functions—Hands-on Exercise

Purpose

The purpose of this lab is to illustrate the creation and use of functions. Functions help promote modular, more organized code. Functions are a major part of the C language in the form of the standard C library, of which printf(), which we have already used, is a member. You may also create your own functions to promote code reuse as well as make your programs more readable and more easily maintained.

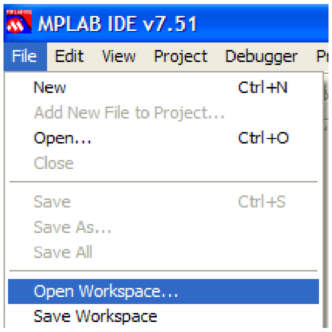
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab08\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab08\...

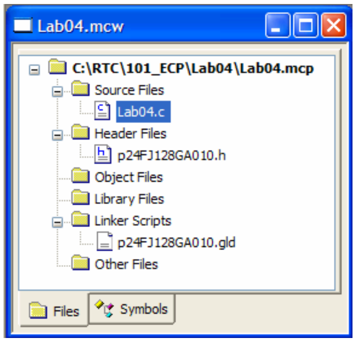
Procedure

On the class PC:
C:\RTC\101_ECP\Lab08\Lab08.mcw



1 Open MPLAB® and select **Open Workspace…** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab08.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

In this lab,

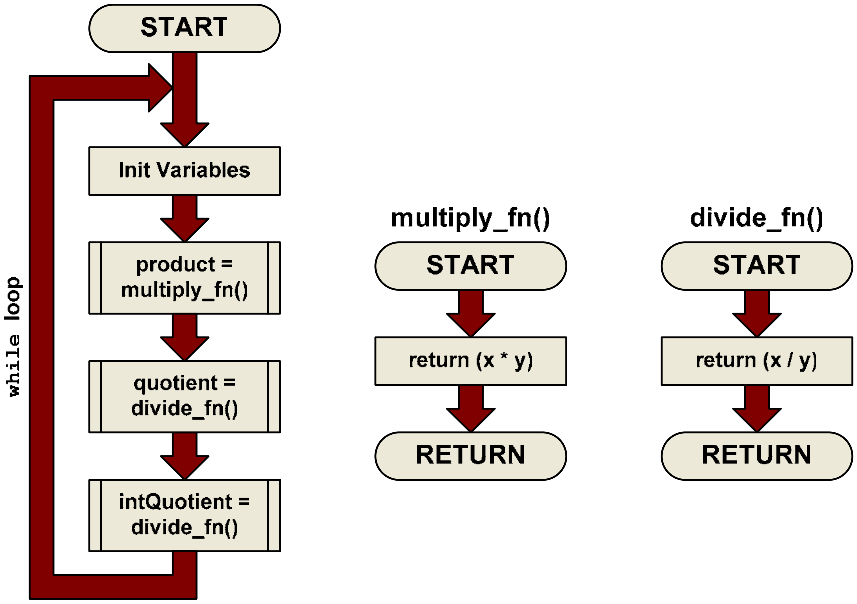
STEP 1:
Write two function prototypes based on the following information:

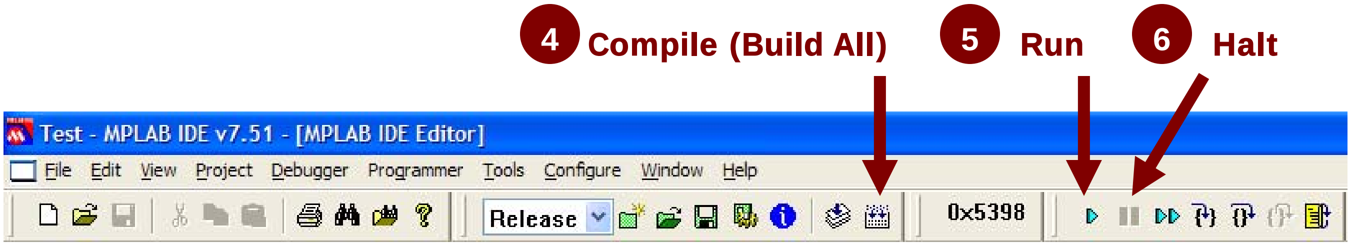
- Function Name: `multiply_function()`
    - Parameters: `int x, int y`
    - Return Type: `int`
- Function Name: `divide_function()`
    - Parameters `float x, float y`
    - Return Type: `float`

STEP 2:
Call the `multiply_function()` and the `divide_function()`.
(a) Pass the variables `intVariable1` and `intVariable2` to `multiply_function()`
(b) Store the result of `multiply_function()` in the variable `product`
(c) Pass the variables `floatVariable1` and `floatVariable2` to `divide_function()`
(d) Store the result of `divide_function()` in the variable `quotient`

STEP 3:
Write the function `multiply_function()`. Use the function prototype you created in step 1 as the function header. In the body, all you need to do is return the product of the two input parameters: `(x * y)`

STEP 4:
Write the function `divide_function()`. Use the function prototype you created in step 1 as the function header. In the body all you need to do is return the quotient of the two input parameters `(x / y)`




4 Compile (Build All)
5 Run
6 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

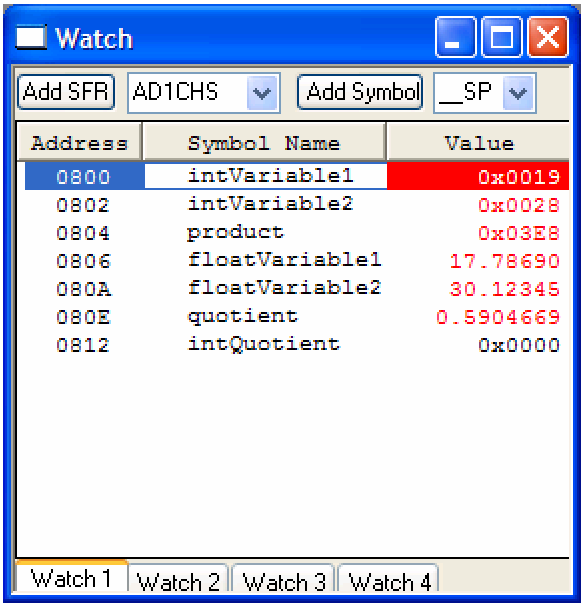
This will run the program in the simulator at full speed.

6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After building, executing and stopping your code, you should see the results of the operations in the watch window as shown at right.



Address	Symbol Name	Value
0800	intVariable1	0x0019
0802	intVariable2	0x0028
0804	product	0x03E8
0806	floatVariable1	17.78690
080A	floatVariable2	30.12345
080E	quotient	0.5904669
0812	intQuotient	0x0000

intVariable1 = 0x19 = 25
intVariable2 = 0x28 = 40
product = 0x3E8 = 1000

floatVariable1 = 17.78690
floatVariable2 = 30.12345
quotient = 0.5904669
intQuotient = 0

Note: intQuotient shows the result of dividing two floating point numbers and storing the result in an integer variable.

Code Analysis

(NOTE: Line numbers correspond to those in the provided solution file.)

Lines 56-59:
STEP 1 was to create the function prototypes for the functions we will write below. The prototypes are required to inform the compiler of the proper format of a function call to these functions so that when it encounters them in the main code before they have actually been defined, it will know that it is not a mistake.

```
      int multiply_function( int x, int y);
      float divide_function( float x, float y );
```

Remember—a function prototype is just the first line (header) of the function definition followed by a semi-colon.

Lines 95-98:
STEP 2 had you make the calls to the functions from within the main routine.

```
      product = multiply_function( intVariable1 , intVariable2 );
      quotient = divide_function( floatVariable1 , floatVariable2 );
```

Since both functions return a value, the proper way to call them is by assigning their results to variables. The variables passed as parameters are defined and initialized higher up in the code.

Lines 119-124:
STEP 3 requires you to write the `multiply_function()` itself. The framework of the function is already there, all you need to do is write the function header (based on the prototype you wrote in step 1) and write the body which only requires you to return the product of the two parameters. The whole function should look like:

```
      int multiply_function(int x, int y)
      {
            return (x * y);
      }
```

The int in front of the header specifies that the function will return an integer type value. The int x and int y in the parameter list allow the function to accept to integer type values as arguments. In the body, we simply return the product of the two parameters.

Lines 140-145:
STEP 4 is almost identical to step 3, but this time you had to write the divide function. The structure of the function is the same:

```
float divide_function( float x, float y )
{
    return (x / y);
}
```

This function returns a floating point value, and takes two floating point parameters. Other than that, and the mathematical operation carried out in the body, it is essentially identical to the `multiply_function()`.

Conclusions

While the functions you created in this exercise were relatively trivial, they do illustrate the syntax and basic mechanism. Functions can be very useful for making code more modular, by taking bocks of code that have a single, well defined purpose and separating them from the main block of code. This has several advantages. First, it makes your code easier to understand and manage. Second, it makes debugging easier because it allows you to have blocks of known good code in separate files (more on this in the next lab). Third, it helps promote code reuse. If you write your functions properly, they can be used over and over again in your future programs.

However, you should be aware that functions can generate extra overhead. While functions can be very useful, and in many cases reduce the amount of code generated, there are situations where overusing functions will result in larger, slower running code. Over time, you will need to develop a sense for when a function makes sense, and when in-line code is a better solution.

Lab 9

Multi-File Projects—Hands-on Exercise

Purpose

The purpose of this lab is to help you understand how to create projects that contain multiple source files. This kind of project structure has numerous advantages, the biggest of which is better organization of programs. It also promotes code reuse because functions that you want to use over and over again may be placed in their own file(s) and then included in any project where you want to use them.

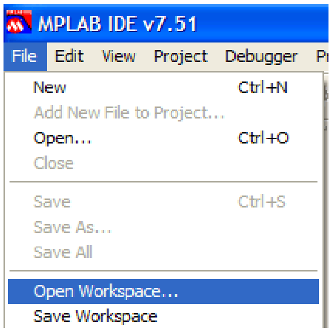
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab09\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab09\...

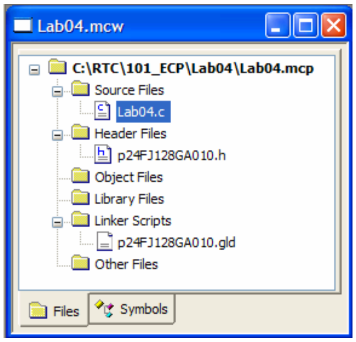
Procedure

On the class PC:
C:\RTC\101_ECP\Lab09\Lab09.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

⚠ If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab09.c source file from the project tree by double clicking on its icon.
You should also open up File1_09.h and File2_09.h, which is where you will be performing the edits in this lab.

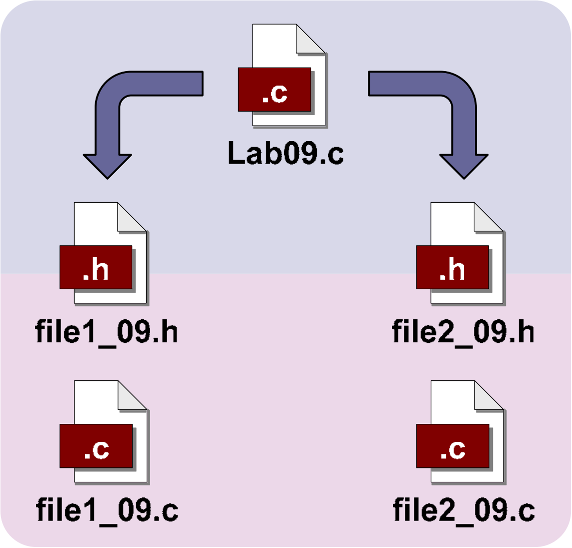
Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

This project is quite a bit different from the others we've worked on so far. We will be dealing with five different files, all of which will be interacting with each other. As before, Lab09.c will contain our main() function, and therefore the program will begin executing from that point. From within the main() function, we will be calling functions that reside in file1_09.c and file2_09.c. The header files associated with file1 and file2 contain the function prototypes that are needed in Lab09.c to be able to call the functions that reside in the separate files.

The files file1_09.c and file2_09.c don't require any editing. In those files, we defined several variables and a couple functions just as we did in Lab8.c. Essentially these files look like any ordinary main file, but without a main() function (a project can only have one of those). Instead, we will be editing the header files, which provide the connection between Lab09.c and the other two C files.

Note that this lab does the exact same thing as Lab08, but the two function definitions and their associated variable definitions have been placed in separate files.



STEP 1a:

Open the file file1_09.h. Add external variable declarations to make the variables defined in file1_09.c available to any C source file that includes this header file. The variables you need to create external definitions for are: intVariable1, intVariable2 and product.

STEP 1b:

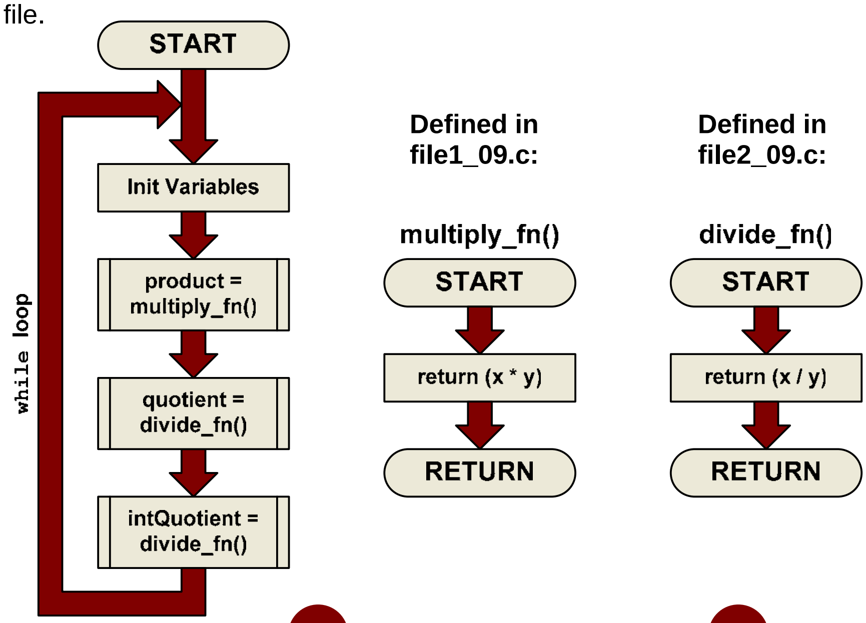
Add a function prototype to make multiply_function() defined in file1_09.c available to any C source file that includes this header file.

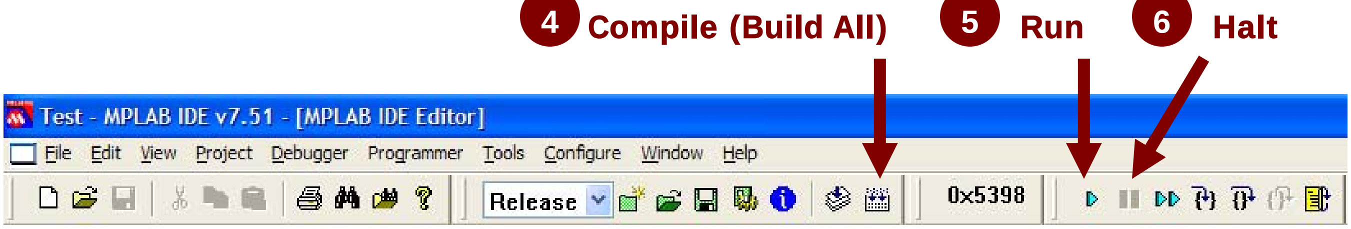
STEP 2a:

Open the file file2_09.h. Add external variable declarations to make the variables defined in file2_09.c available to any C source file that includes this header file. The variables you need to create external definitions for are: floatVariable1, floatVariable2, quotient and intQuotient.

STEP 2b:

Add a function prototype to make divide_function() defined in file2_09.c available to any C source file that includes this header file.




4 Compile (Build All)
5 Run
6 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button. This will run the program in the simulator at full speed.

6 Click on the **Halt** button. This will stop execution so that we may examine the variables and their values.

Results

The results for Lab09 should be identical to the results for Lab08 (see page 46).

Code Analysis

The programs of Lab08 and Lab09 are identical, but we moved the multiply_function() and its associated variables into file1_09.c and we moved divide_function() and its associated variables into file2_09.c. Nothing was changed with respect to their syntax. The new element is the set of header files used to interface the main file with the other two files.

file1_09.h Lines 23-28:
STEP 1a asks us to write external declarations for the variables defined in file1_09.c. In that file, we find three variables:

```
int intVariable1 = 0;
int intVariable2 = 0;
int product = 0;
```

In order to make them available to other source files, they must be declared as extern where they will be used. The easiest way to provide this is to put the extern declarations into a header file that may be included in any source file that will use these variables. All you need to do is duplicate the variable declarations from file1_09.c and put the extern keyword in front of them (note—you cannot initialize an extern variable declaration—only the actual definition may use an initializer):

```
extern int intVariable1;
extern int intVariable2;
extern int product;
```

file1_09.h Lines 39-40:
STEP 1b asks you to provide a function prototype to make multiply_function() available to other files. This function prototype will be identical to the one you created in Lab08:

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

file2_09.h Lines 23-28:
STEP 2a is similar to step 1a, but this time we need to make the variables in file2_09.c available to other files. The variables in file2_09.c are:

```
float floatVariable1 = 0;
float floatVariable2 = 0;
float quotient = 0;
int intQuotient = 0;
```

And just like step 1a, we need only add the extern keyword in front of them when they are added to the header file:

```
extern float floatVariable1;
extern float floatVariable2;
extern float quotient;
extern int intQuotient;
```

file2_09.h Lines 41-42:
STEP 2b, we need to make divide_function() available to other files by adding a function prototype here. Again, this is just like the one you created in Lab08.

```
float divide_function(float x, float y );
```

Conclusions

Multi-file projects take the concept of functions a step further, by allowing further organization and separation of functionality within a program. Separate files allow you to group related functions and variables in such a way that they may be used among several different programs. Using multiple files incurs no additional overhead, so you can feel free to use them whenever they make sense.

Lab 10

Arrays—Hands-on Exercise

Purpose

The purpose of this lab is to help you understand how to create and use arrays. The main concepts include defining and initializing arrays, passing array elements to functions and assigning function results to array elements. The code will also illustrate how to print out elements of an array to the Sim Uart1 I/O window.

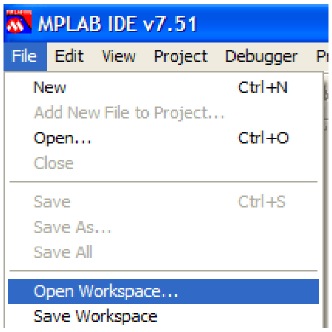
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab10\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab10\...

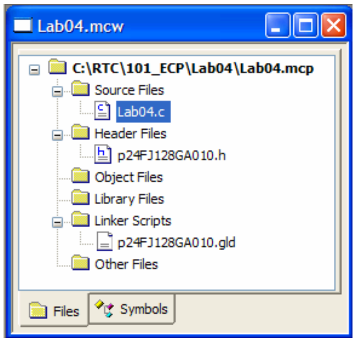
Procedure

On the class PC:
C:\RTC\101_ECP\Lab10\Lab10.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab10.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

In this lab, some of the code has been placed in a separate file to keep things simple where you need to make your edits. Specifically, the printArray() and print2dArray() functions have been placed in the file PrintArray.c.

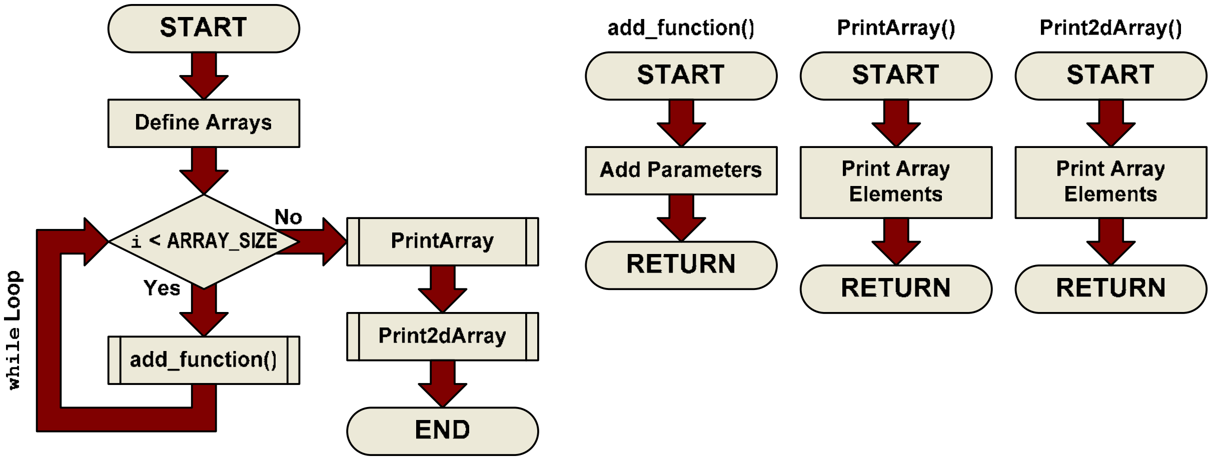
All of the edits in this lab will be made in Lab10.c

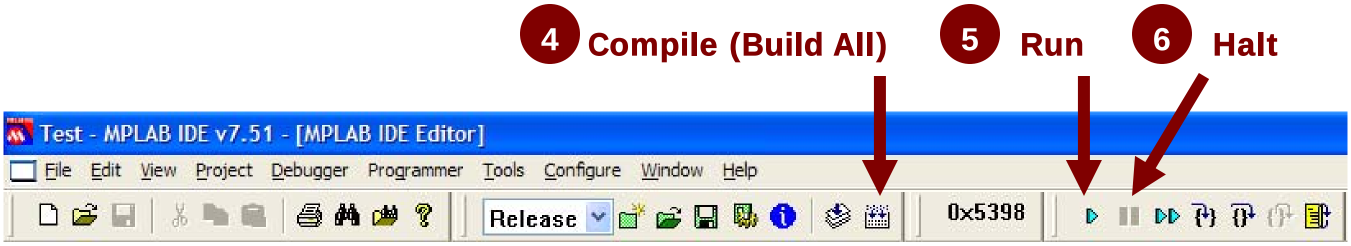
STEP 1:
Create two initialized arrays with 10 elements each named array1 and array2 (you may use the pre-defined constant ARRAY_SIZE as part of the array declaration). The arrays should be initialized with the following values:

- array1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- array2: 9, 8, 7, 6 ,5, 4, 3, 2, 1, 0

Note: the elements are all of type int.

STEP 2:
Pass the two arrays you declared above (array1 and array2) to the function add_function() (see its definition below). Store the result of the function in the array result[]. The idea here is to add each corresponding element of array1 and array 2 and store the result in result[]. In other words, add the first element of each array and store it in the first element of result[], then add the second elements, and so on. Take advantage of the counter variable i to make this happen.





4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

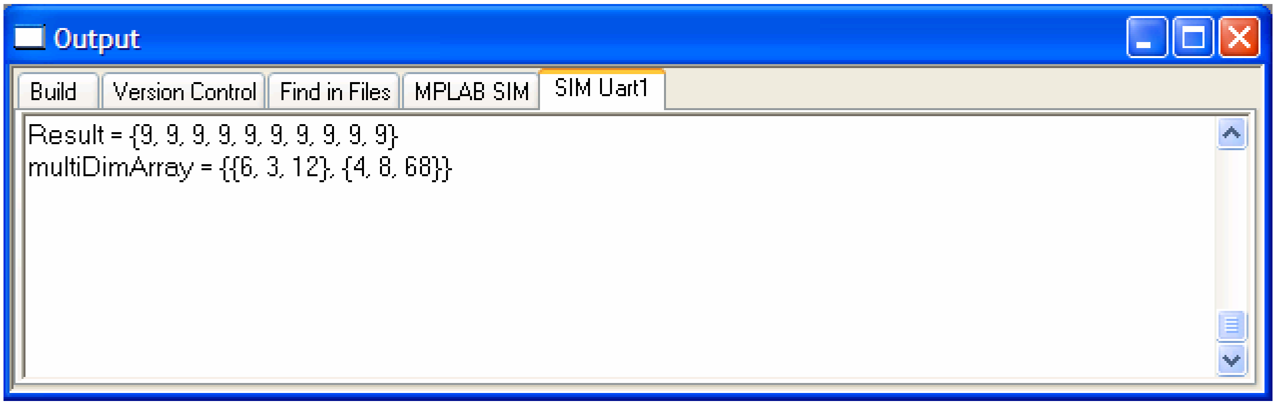
This will run the program in the simulator at full speed.

6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After building and running your code, you should see the following results in the SIM Uart1 window:



```
Result = {9, 9, 9, 9, 9, 9, 9, 9, 9, 9}
multiDimArray = {{6, 3, 12}, {4, 8, 68}}
```

Code Analysis

Lines 47-50:
STEP 1 asked you t o create two initialized arrays, and gave you the size in the form of a constant (ARRAY_SIZE) and the data to initialize them with. This code should be very similar to some of the syntax examples in the presentation:

```
int array1[ARRAY_SIZE] = {0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9};
int array2[ARRAY_SIZE] = {9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 , 0};
```

So, both the arrays look slimier. To initialize them as part of their declaration, all you needed to do was follow the declaration by an equals sign and a comma delimited list of data enclosed in curly braces.

Lines 83-84:
At this point, we are inside a while loop, with an incrementing variable i which is checked against the constant ARRAY_SIZE at each pass of the loop. The idea here was to pass the corresponding elements of each array to the add_function() and store the result in the array result[]. To do this, we can use the counter variable i as the array index for each pass of the loop:

```
          result[i] = add_function(array1[i], array2[i]);
```

In the context of the loop, the following operations will take place:

result[0] = array1[0] + array2[0]
result[1] = array1[1] + array2[1]
result[2] = array1[2] + array2[2]
result[3] = array1[3] + array2[3]
…
result[9] = array1[9] + array2[9]

Conclusions

Here, you have learned how to create and use arrays. Frequently, arrays are manipulated within loops where the same operation needs to be carried out on a series of related variables, as we have seen here. The array index may be a variable, which gives us tremendous flexibility for accessing arrays based on some conditions we setup in our programs.

Arrays may also be used in a lookup table like scheme, where a variable is used as an offset into the table (via the array index) to retrieve a particular value from the list (array values).

Arrays are also used to implement text strings, though we will not put too much emphasis on the topic in this course since text strings are rarely used in embedded systems. They are typically only found in systems incorporating sophisticated user interfaces with some sort of text capable display (LCD, VFD, PC interface, etc.)

Lab 11

Pointers—Hands-on Exercise

Purpose

This lab will serve as your first introduction to pointers. For many programmers, this is one of the more difficult concepts to grasp. The exercises presented here are by necessity very simplistic in order to help you get a firm grasp on the syntax associated with pointers. The application code presented here isn't necessarily showing the best use of pointers, but it will clearly illustrate how they work, which will make it easier for you to implement them in more complex situations where they are most useful. So for the moment, if you can avoid the "why would anyone do this?" question, and focus on the syntax and mechanism of pointers, in the end you will be much better equipped to make use of them in your own code when they are required.

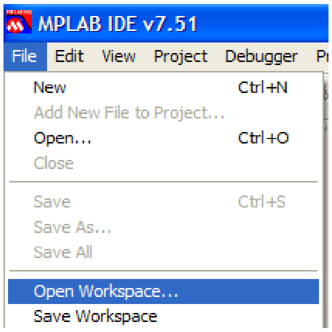
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab11\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab11\...

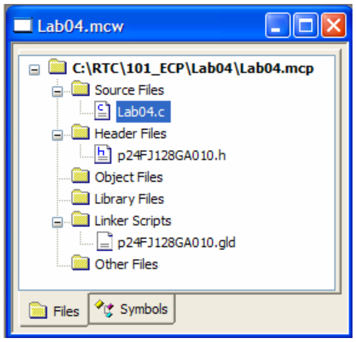
Procedure

On the class PC:
C:\RTC\101_ECP\Lab11\Lab11.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

⚠ If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab11.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

STEP 1:
Initialize the pointer p with the address of the variable x.
Hint:: Use the unary & operator.

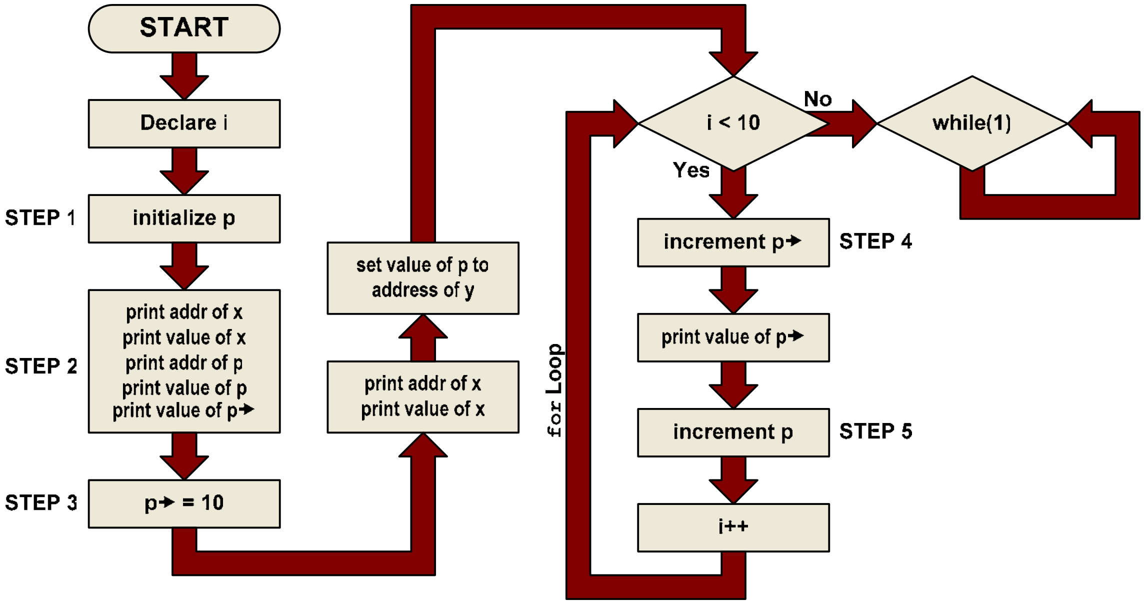
STEP 2:
Complete the following printf() functions by adding in the appropriate arguments as described in the control string. NOTE: This program will not build unless the following 5 lines of code are completed and the comments are removed from within the printf() functions.
(1) Address of the variable x - use address of operator
(2) Value of the variable x
(3) Address of the pointer p itself (not what p points to) - use address of operator
(4) Value of the pointer p (address of what p points to)
(5) Value pointed to by p (value stored in location p points to) - dereference the pointer

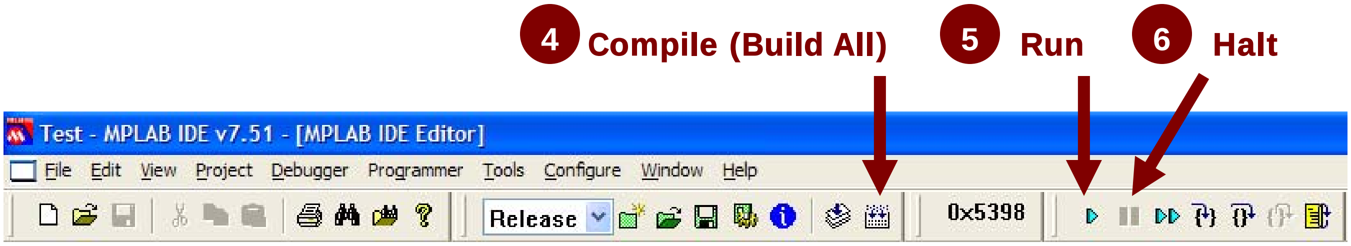
STEP 3:
Write the integer value 10 to the location that p points to. - dereference the pointer

STEP 4:
Increment the value that p points to. - dereference the pointer

STEP 5:

Increment the pointer p so that it points to the next item. Don't forget operator precedence—use parentheses if necessary.





4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

This will run the program in the simulator at full speed.

6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After building and running your code, you should see the following results in the SIM Uart1 window:

```
The variable x is located at address 0x8AA
The value of x is 5
The pointer p is located at address 0x8D8
The value of p is 0x8AA
The value pointed to by *p = 5
The variable x is located at address 0x8AA
The value of x is 10
```

```
y[0] = 1
y[1] = 2
y[2] = 3
y[3] = 4
y[4] = 5
y[5] = 6
y[6] = 7
y[7] = 8
y[8] = 9
y[9] = 10
```

Code Analysis

(NOTE: Line numbers correspond to those in the provided solution file.)

Lines 34-36:
Here, three variables are created for you:
int x, an ordinary integer, which is initialized with a value of 5. This variable will be assigned a location in RAM
int y[10], an array of integers, initialized with values from 0 to 9. This variable will be assigned to a series of 10 sequential locations in RAM.
int *p, a pointer to an integer. The variable p will be assigned an address in RAM, and at that address, it will store the address of whatever variable it points to. p is not initialized, so for the moment, it has a value of NULL (i.e. its value is zero).

Line 52:
A local variable (local to main()) is declared. We will use this variable as a counter in the for loop below.

Line 59:
Here, you were instructed to assign the address of x to the pointer to p. This is done by setting the variable p, equal to the address of x, using the address of '&' operator:

```
p = &x;
```

Because we are writing the address of a variable into the pointer, we simply use the variable p, since that is where the address will be stored (no * is used in this case). In other words, the address of x is stored at the address of p.

Lines 66-70:
Here you were instructed to complete the printf() functions, based on the text description in their control strings.

(1) To complete this line, we need to refer to the address of the variable x, this is done just like we did on line 59 by using the address of operator:
`printf("The variable x is located at the address 0x%X\n", &x);`

(2) For this line, we need to refer to the value of the variable x. This is done by simply using the variable x by itself:
`printf("The value of x is 0x%X\n", x);`

(3) Now, we are going to work with the pointer variable. The intention of this line is to print out the address of the pointer variable p itself. This can be done the same way as we referred to the address of x:
`printf("The pointer p is located at address 0x%X\n", &p);`

(4) Next, we need to print out the value contained in p. This is the address of the object that p points to. To refer to a pointer's value, you just use the pointer variable by itself, just like we did for x:
`printf("The value of p is 0x%X\n", p);`

(5) Finally, we are asked to print the value pointed to by p. To do this, we need to use the dereference operator '*' with the pointer p (the control string provided a hint):
`printf("The value pointed to by *p is 0x%X\n", *p);`
The dereference operator allows us to retrieve whatever value is at the location that p is pointing to.

Line 77:
On this line, you were instructed to write the value 10 to the location pointed to by p. To do this, we need to use the dereference operator as we did in the last part of the previous step:

```
*p = 10;
```

Line 82:
Here, we reassign the pointer to point to the first element of the array y[]. We can do this because the only requirement for p is that it points to an integer. Since the elements of y[] are all integers, it is just as valid to have p point to one of y[]'s elements as it is to have it point to x.

Line 93:
We are now inside the for loop, where we will go through every element of the array y[] and print out its value. Instead of using the loop counter as the array index and using normal array notation to do this, we are going to use a pointer to iterate through each element of the array. This is a great illustration of the fact that arrays are merely simplified notation for pointers. Our first task is to increment the value that p points to. Due to operator precedence rules, we need to use parentheses to make this work properly:

```
(*p)++;
```

This syntax means that we first dereference the pointer, then perform the increment, so that the item pointed to by p is what gets acted on by the increment operator.

Line 95:
This line simply prints out the value contained in the array element that is currently pointed to by p. Note the use of the *p as the item in the argument list.

Line 100:
Now, we need to increment the pointer itself. To do this, we simply operate on the pointer variable itself:

```
p++;
```

Because of the way the compiler handles pointer arithmetic, this operation may or may not increment the value in p by 1. In this particular situation, where the 16-bit PICs have byte addressable data memory, the value in p will actually be incremented by 2 since integer type variables occupy two bytes.

Conclusions

While this code doesn't perform any particularly useful tasks, it does illustrate the functionality of the pointer mechanism in C. The main idea was for you to understand what syntax to use in which situations. Hopefully this simple example made it easy to see what specifically is going on with all the ways you can work with pointers and variable addresses. From this you should have learned that everything, including pointers themselves have an address, and that a pointer of a particular type can point to anything of that same type, whether it be an individual variable or an array. Similarly, you should now see that pointers and arrays are very closely related, but that pointers are much more flexible, though they have a more complicated syntax. It should also be noted that pointers are one of the more difficult concepts to grasp in C, and that as a result, most programming errors are due to inappropriate use of pointers. So, the better you understand this section, the better C programmer you will be.

Lab 12

Pointers, Arrays, and Functions—Hands-on Exercise

Purpose

This lab will continue and expand upon the lessons of lab 11. Here, we will look at how pointers and arrays may be passed to functions, and the associated syntax you must use at every step, both inside and outside the function. It will also demonstrate further the relationship between arrays and pointers, and how they can be used interchangeably in some cases.

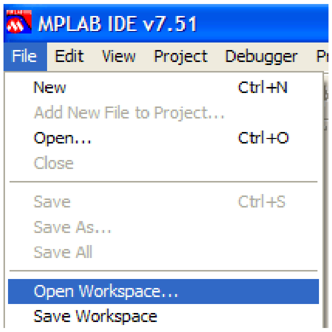
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab12\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab12\...

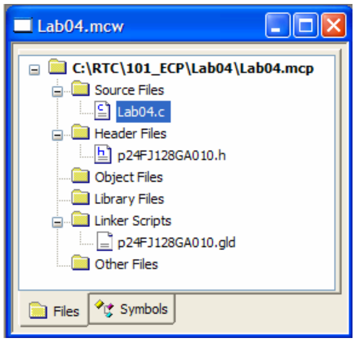
Procedure

On the class PC:
C:\RTC\101_ECP\Lab12\Lab12.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab12.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

STEP 1:
Pass the variable x to the function twosComplement() such that the value of x itself may be changed by the function. Note: the function expects a pointer (address) as its parameter.

```
void twosComplement(int *number) {…}
```

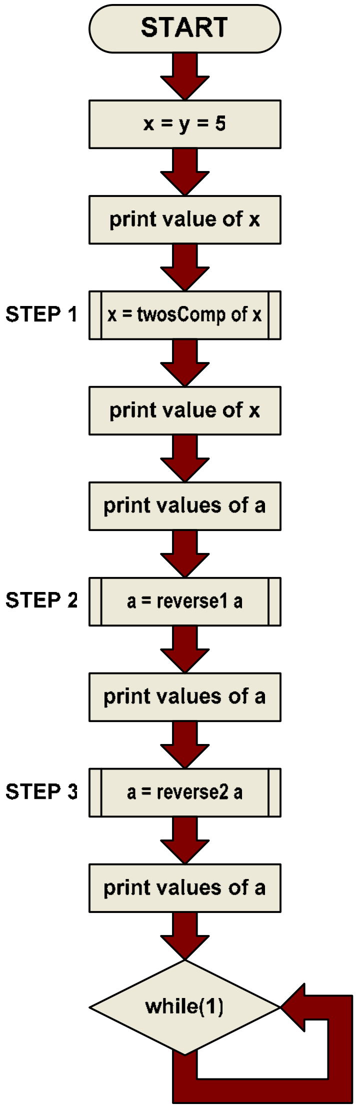
STEP 2:
Pass the array 'a' to the function reverse1(). Use the constant ARRAY_SIZE for the second parameter. See definition of function reverse1() below.

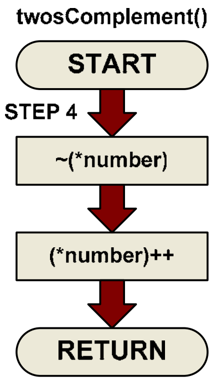
```
void reverse1(int numbers[], const int SIZE) {…}
```

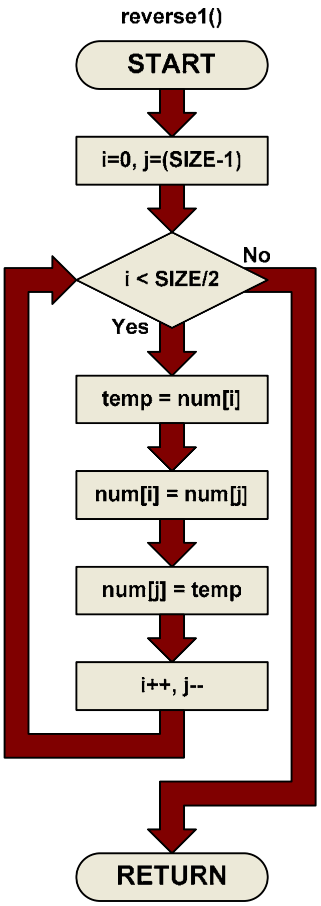
STEP 3:
Pass a pointer to array 'a' to the function reverse2(). Use the constant ARRAY_SIZE for the second parameter. See the definition of the function reverse2() below. Hint: You do not need to define a new pointer variable to do this.

```
void reverse2(int *numbers, const int SIZE) {…}
```

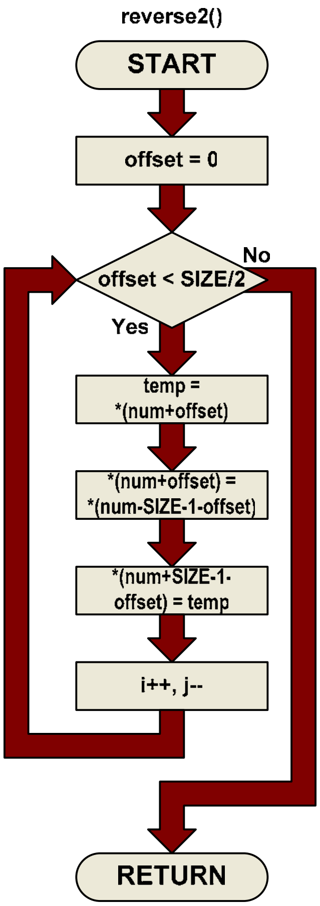
STEP 4:
Complete the function header by defining a parameter called 'number' that points to an integer (i.e. accepts the address of an integer variable).



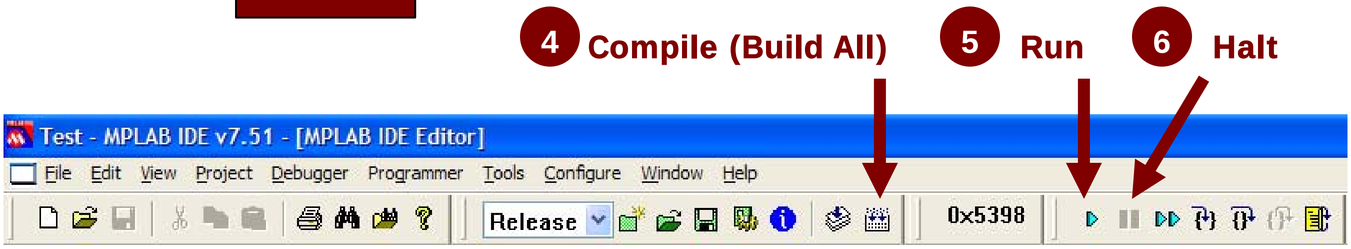




reverse1() uses an array parameter



reverse2() uses a pointer parameter


4 Compile (Build All)
5 Run
6 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

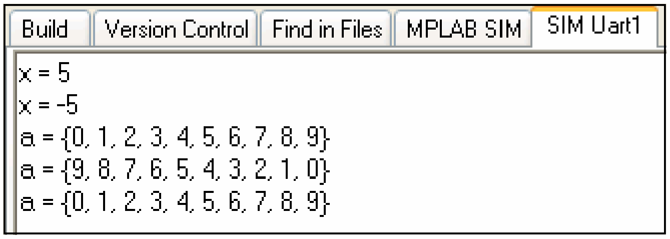
This will run the program in the simulator at full speed.

6 Click on the **Halt** button.

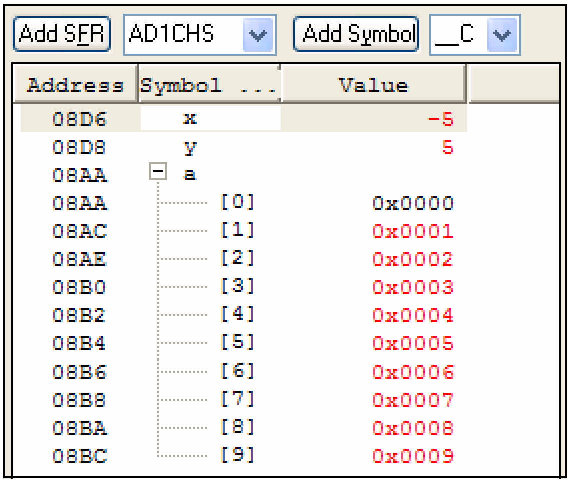
This will stop execution so that we may examine the variables and their values.

Results

After running the code, you should see the following results in the Sim Uart1 window and the watch window:



```
x = 5
x = -5
a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
a = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```



Address	Symbol ...	Value
08D6	x	-5
08D8	y	5
08AA	⊟ a	
08AA	[0]	0x0000
08AC	[1]	0x0001
08AE	[2]	0x0002
08B0	[3]	0x0003
08B2	[4]	0x0004
08B4	[5]	0x0005
08B6	[6]	0x0006
08B8	[7]	0x0007
08BA	[8]	0x0008
08BC	[9]	0x0009

The twosComplement() function simply manipulates the binary form of an integer into its negative valued representation. Obviously, it would be much easier to just put a minus sign in front of a variable. This function was used only to demonstrate the parameter passing mechanism.
The reverse1() and reverse2() functions both reverse the order of elements in an array, but the first one takes an array name as a parameter and the second one takes a pointer to the first array element as its parameter. In either case, the end result is the same, but accomplished differently.

Code Analysis

Line 62:
In order to pass a variable to a function so that the function can modify the original variable, we need to pass the variable by reference. In other words, we need to pass a pointer to the variable to the function instead of the variable itself (pass by value).

```
twosComplement(&x);
```

Line 73:
When you pass an array to a function, you need only pass the name of the array (without the index brackets). Unlike normal variables, arrays are always passed by reference. The name of an array is equivalent to a pointer to its first element, and is treated as such in many cases.

```
        reverse1(a, ARRAY_SIZE);
```

Line 84:
This time, we are instructed to pass a pointer to the array's first element to the function reverse2(). Since an array's name is the same thing as a pointer to its first element (the array name without its index brackets represents the address of the first element), we can simply pass the name to the function as we did in the previous step.

```
        reverse2(a, ARRAY_SIZE);
```

Although the code of reverse2() is quite different from the code in reverse1(), it still works the same way when you pass the array name to it.

Line 106:
To pass an address parameter to a function, that parameter must be declared as a pointer. This declaration looks like an ordinary pointer declaration, but it occurs within the parameter list of the function:

```
void twosComplement(int *number)
```

So, the parameter *number is expecting to be passed an address of an int variable so that the function can directly manipulate that variable rather than just receive its value.

Conclusions

One of the most common use of pointers is to pass function parameters by reference rather than by value, so that the function can operate directly on the variable being passed to it, rather than simply receiving a copy of the value contained in the variable. To pass a variable by reference to a function, the function parameter must be declared as a pointer, and the value passed to the function must be a pointer itself, or a variable preceded by the address of operator ‘&’. Within the function itself, the dereference operator ‘*’ must be used to access the actual variable that was passed to the function.

You have also seen that arrays and pointers are even more closely related than shown in lab 11. An array’s name without the index brackets is the equivalent to a pointer to the first element of the array. An array’s name can in many cases be used where a pointer to the type of the array’s elements is expected—particularly in function calls, where the array parameter would be passed by reference in any case.

Lab 13

Function Pointers—Demo

Purpose

This demo provides a working example of function pointers in action. Function pointers are not frequently used in C programming (perhaps due to their strange syntax), but can be extremely useful in some circumstances.

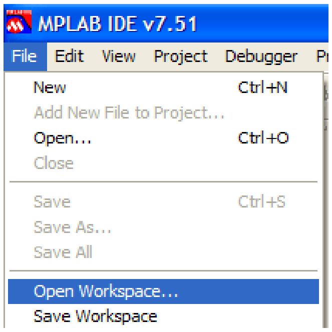
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab13\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab13\...

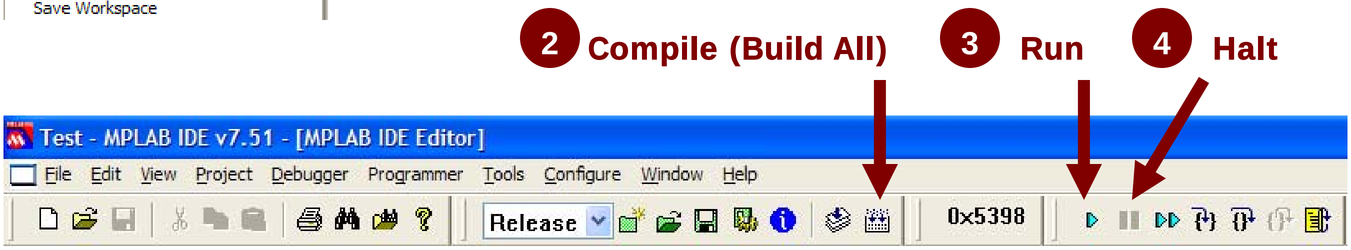
Procedure

On the class PC:
C:\RTC\101_ECP\Lab13\Lab13.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

⚠ If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.


2 Compile (Build All)
3 Run
4 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button. This will run the program in the simulator at full speed.

4 Click on the **Halt** button. This will stop execution so that we may examine the variables and their values.

What just happened?

As was done earlier in the class, we opened a pre-configured MPLAB® workspace with a complete, working program. We then compiled the code and ran it long enough for it to complete its task. This program uses a function pointer to pass the address of a mathematical function to another function that will compute its integral.

The integral example was adapted from one published on Wikipedia at: http://en.wikipedia.org/wiki/Function_pointer. The integral function takes three parameters: the upper and lower bounds of the integral, and the address of the function that it is to evaluate. The function's header looks like:

```
float integral(float a, float b, float (*f)(float))
```

Note that the third parameter is defined as a function pointer. When we call this function, we only need to provide the name of the function we want to integrate. For example:

```
y2 = integral(0, 1, xsquared);
```

The function xsquared() is a simple mathematical function defined as:

```
float xsquared(float x)
{
      return (x * x);
}
```

There are other functions that may be passed to the integral() function as well.

Results

The program evaluates the integral of three functions: y=x, y=x^2 and y=x^3. After running the program you should see the following printed out in the Sim Uart1 window:

```
y1 = integral of x dx over 0 to 1 = 0.500000
y2 = integral of x^2 dx over 0 to 1 = 0.335000
y3 = integral of x^3 dx over 0 to 1 = 0.252500
```

Code Analysis

Lines 55, 61, and 67:

These three lines make calls to the integral() function. Each one passes a different function's address to the integral() function for evaluation. The address of a function is represented by the function's name alone (no parentheses or parameters).

```
      y1 = integral(0, 1, justx);
      y2 = integral(0, 1, xsquared);
      y3 = integral(0, 1, xcubed);
```

Lines 80-83:

This function justx() simply returns the value of x (y = x)

```
float justx(float x)
{
      return x;
}
```

Lines 92-95:

This function xsquared() simply returns the value of x^2 (y = x^2)

```
float xsquared(float x)
{
      return (x * x);
}
```

Lines 104-107:

This function xcubed() simply returns the value of x^3 (y = x^3)

```
float xcubed(float x)
{
      return (x * x * x);
}
```

Lines 119-132:

This is the integral() function which will evaluate the integral of any mathematical function passed to it over the range specified by the lower bound a and the upper bound b. The third parameter of the function header is a function pointer. It expects to receive the address of a function, which may be passed simply as the name of a function.

```
float integral(float a, float b, float (*f)(float))
{
    float sum = 0.0;
    float x;
    int n;

    //Evaluate integral{a,b} f(x) dx
    for (n = 0; n <= 100; n++)
    {
        x = ((n / 100.0) * (b-a)) + a;
        sum += (f(x) * (b-a)) / 101.0;
    }
    return sum;
}
```

The algorithm used to evaluate the integral is beyond the scope of this course. But it should be noted that the function name that is passed to this function is accessed via its parameter name f. For example, the line that states:

```
        sum += (f(x) * (b-a)) / 101.0;
```

invokes the function passed via f, and passes the parameter x to it. So, when the function xsquared() is passed, then f(x) evaluates xsquared(x), where x is a local variable defined within integral() and is defined in the line immediately above.

Lab 14

Structures—Hands-on Exercise

Purpose

This lab will help illustrate the use of structures in C. The code is a bit more complex than previous programs, but it will help to show how structures can simplify what might otherwise be very complicated code. In this code, we perform circuit power calculations using two methods. The first uses simple structures while the second uses a structure of structures. You will also see how pointers to structures may be used to copy an entire structure from one variable to another.

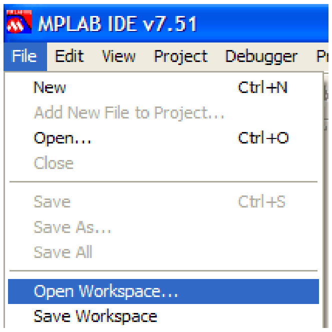
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab14\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab14\...

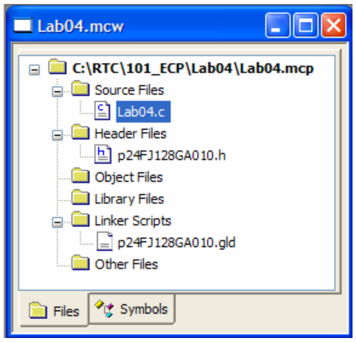
Procedure

On the class PC:
C:\RTC\101_ECP\Lab14\Lab14.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

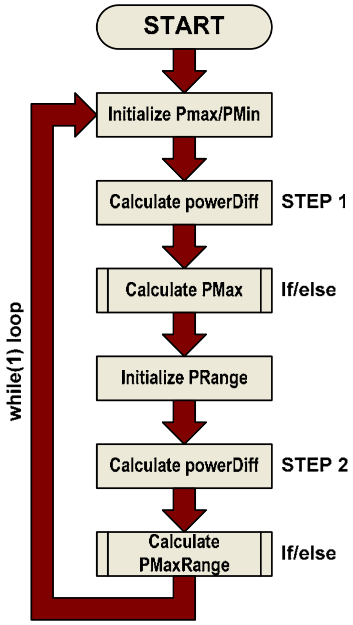
If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



2 If it isn't already visible in the workspace, open the Lab14.c source file from the project tree by double clicking on its icon.

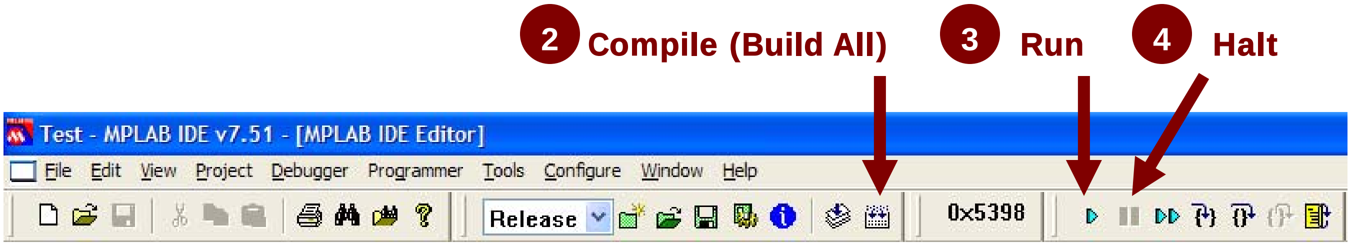
Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments



STEP 1:
Calculate the difference between the minimum and maximum power in circuit 1 using the individual power structures (i.e. the variables PMax1 and PMin1). Algebraically, we want to compute:
Pdiff = (Vmax * Imax) - (Vmin * Imin)
(HINT: Look at the lines below if you are having trouble)

STEP 2:
Calculate the difference between the minimum and maximum power in circuit 1 using the structure of structures (i.e. the variable PRange1). Algebraically, we want to compute:
Pdiff = (Vmax * Imax) - (Vmin * Imin)
(HINT: Look at the lines below if you are having trouble)


2 Compile (Build All)
3 Run
4 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button.

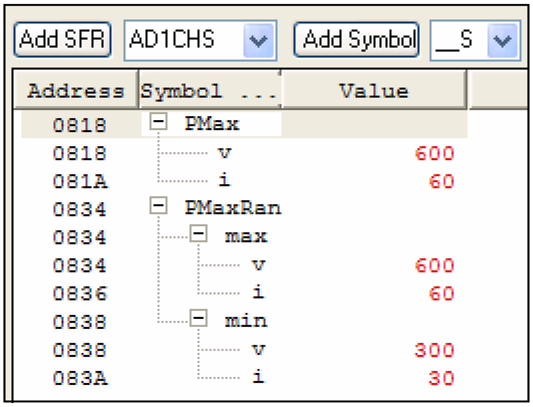
This will run the program in the simulator at full speed.

4 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

The program calculates the power in three different circuit, and then determines which circuit has the highest power. After running the program, you should see the following results shown at right in the watch window.
Note that MPLAB® presents structure variables such that you can expand and collapse them to either show or hide the individual members of the structure.



```
Address  Symbol ...        Value
0818     ⊟ PMax
0818         v               600
081A         i                60
0834     ⊟ PMaxRan
0834       ⊟ max
0834           v             600
0836           i              60
0838       ⊟ min
0838           v             300
083A           i              30
```

Code Analysis

Lines 32-35:
The first of two different structures is defined here. The `power` structure is used to hold the minimum or maximum voltage and current measurements for a circuit. There will be two variables of type `power` declared for each circuit—one to hold the maximum values, the other to hold the minimum values.

Lines 37-40:
This second structure is actually a structure of structures. It is designed to hold two variables of type `power`. So, each `range` structure can hold the maximum and minimum power measurements for a circuit. Since we are creating members of type `power` within the `range` structure, the `power` structure needed to be declared first.

Lines 46-49:
These four variables will be used to store the results of the power calculations we will perform later in the program. They are all ordinary long integer type variables.

Lines 55-66:
These lines declare variables of the two structure types. The first group of variables are all of type `power`, and we will use them in the first part to show how the power can be calculated using structures with members of C's built-in data types. The second group are all of type range, and will be used in the second part of the program to show how `power` can be calculated using a structure with members that are structures themselves.

Lines 68-69:

Here, we declare two pointers. One points to a structure variable of type `power`, and the other points to a structure variable of type `range`. These will be used later in the program to first point to the particular circuit's power structure with the maximum value. It will then be used to copy those structures' values into the two maximum power structure variables `PMax` and `PMaxRange`.

Lines 101-115:

At this point of the program, we are inside the main loop and need to initialize the variables that we will be using. In a real application, we probably would obtain these values by sampling the circuits with an analog to digital converter. However, since we are working in the simulator, and have no hardware connected, we will simply make up some values and assign them to the variables.

Line 127:

STEP 1 requires that you calculate the difference between minimum and maximum power in circuit 1 using the variables of type `power`. This can be done just like it is done for circuits 2 and 3 on lines 128-129:

```
powerDiff1 = (PMax1.v * PMax1.i) - (PMin1.v * PMin1.i);
```

The variables PMax1 and PMin1 each have two members: v and i, which represent the voltage and current measurements respectively. So, to calculate the maximum power, we simply need to multiply the v and i members of PMax1 together. The minimum power is calculated in the same way. Then, to get the difference, we subtract the minimum calculation from the maximum calculation. The result is then stored in powerDiff1.

Lines 135-150:

This block of code determines which of the three circuits have the greatest difference between maximum and minimum power. It first checks to see which of powerDiff1 and powerDiff2 is greater, and assigns the larger one to maxPowerDiff. Next, it checks to see which of maxPowerDiff and powerDiff3 is greater. If maxPowerDiff is greater, nothing further is done. If powerDiff3 is greater, it is assigned to maxPowerDiff.
Also, in each of the above steps, the address of the structure containing the maximum power of the one that has the greatest difference is assigned to the pointer pPower.

Line 156:

The pointer pPower that we initialized above is now used to copy the maximum power value into the structure variable PMax. When using a pointer in this fashion, the values of all of the members of the structure it points to are copied into the variable on the left of the assignment operator.

Lines 167-181:

Just like lines 101-115 above, we need to initialize the variables we plan on using as if we were actually obtaining these from an analog to digital converter.

Line 193:

STEP 2 calculates the same thing as line 127 above, but this time it uses the structure of structures variables. The concept is the same, but the syntax is now a bit different because we need to reference two levels of structure members:

```
powerDiff1 = (PRange1.max.v   * PRange1.max.i) - (PRange1.min.v * PRange1.min.i);
```

Since the range of power is stored in `PRange1` as two `power` type members, we need to add an extra level of structure member references to our calculation. For example, PRange1.max.v refers to the voltage member of the structure variable max, which itself is a member of the structure PRange1. In total, PRange1 stores four values.

Lines 201-216:

This block determines which circuit has the greatest power difference, just as was done on lines 135-150.

Line 223:

This line does essentially the same thing as line 156.

Lab 15

Arrays of Structures—Hands-on Exercise

Purpose

This lab will introduce you to the syntax used to work with arrays of structures. For this exercise, we have defined a structure to hold the real and imaginary parts of a complex number. You will then need to modify the real and imaginary parts of each element of an array of complex numbers.

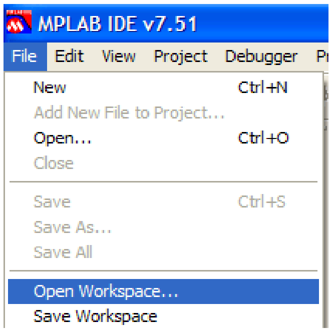
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab15\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab15\...

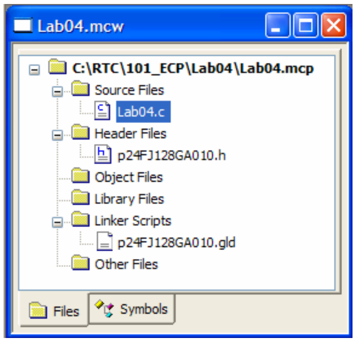
Procedure

On the class PC:
C:\RTC\101_ECP\Lab15\Lab15.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



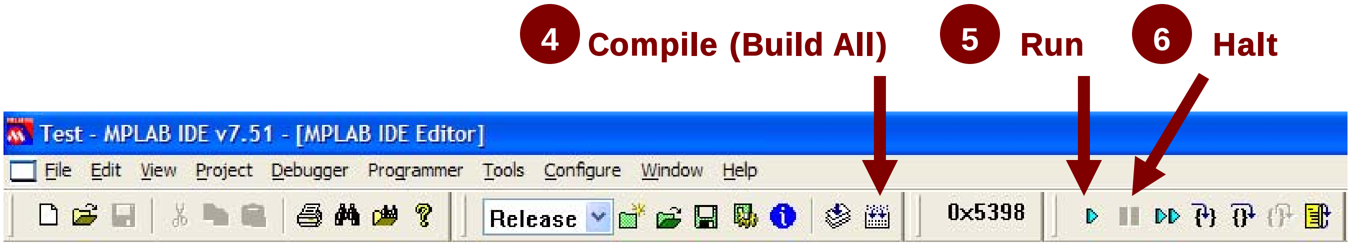
2 If it isn't already visible in the workspace, open the Lab15.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 Edit source code as instructed in the comments

STEP 1:
Multiply the real (re) real part of each array element by 10. (HINT: Use *=)

STEP 2:
Multiply the imaginary (im) part of each array element by 5 (HINT: Use *=)


4 Compile (Build All)

5 Run

6 Halt

Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

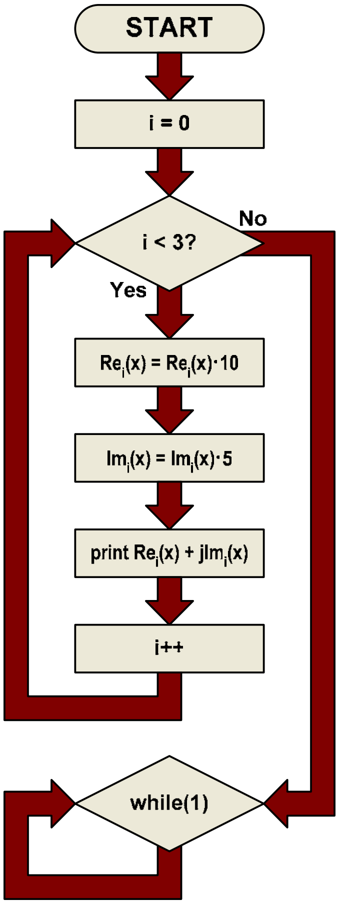
4 Click on the **Build All** button.

5 If no errors are reported, click on the **Run** button.

This will run the program in the simulator at full speed.

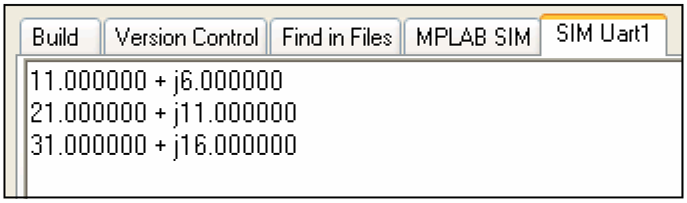
6 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.


START
i = 0
i < 3?
No
Yes
$Re_i(x) = Re_i(x)\cdot 10$
$Im_i(x) = Im_i(x)\cdot 5$
print $Re_i(x) + jIm_i(x)$
i++
while(1)

Results

After successfully building and running your code, you should see the following in the Sim Uart1 window:



```
11.000000 + j6.000000
21.000000 + j11.000000
31.000000 + j16.000000
```

Code Analysis

Lines 28-31:
This is the type declaration for the complex number structure. There are two members, representing the real (`re`) and imaginary (`im`) parts, and the type name is `complex`.

Line 36:
This is a variable declaration. We are creating an array of structures of type complex. The array has three elements, and we are initializing the array as we are declaring it. Note that the syntax we use to initialize the array is essentially the same as initializing a multidimensional array. So, in a more algebraic form, our array now looks like this:
x[0] = 1.1 + j1.2
x[1] = 2.1 + j2.2
x[2] = 3.1 + j3.2

Line 65:
At this point, we are now inside the loop, and will be accessing a complex number on each pass. STEP 1 has us multiplying the real part of the current array element by 10. The easiest way to do this is to use the *= operator. The structure member may be accessed by using the dot notation with the array variable and its index:

```
      x[i].re *= 10;
```

Line 72:
STEP 2 has us performing a similar operation to step 1, but this time, we will be multiplying the imaginary part by 5:

```
      x[i].im *= 5;
```

Line 77:
This line prints out the recently modified complex numbers in an algebraic format:

```
          printf("%f + j%f\n", x[i].re, x[i].im);
```

Since the complex number's members are both floating point types, we need to use %f as our placeholder/formatting characters in the printf() control string. Then, the arguments used are the same as those used on Line 65 and Line 77 to specify a structure member that is part of an array element.

Lab 16

Unions—Hands-on Exercise

Purpose

This lab will help you to understand how to work with unions. The code is very short and performs no practical function, but it will enable you to work with a union variable and observe how it makes use of data memory when values are written to its members.

Note that you will be using the simulator a bit differently this time. Rather than simply running and stopping your code to see the results, you will be setting breakpoints at three points in your code, and stopping at each of them to observe the data in the watch window.

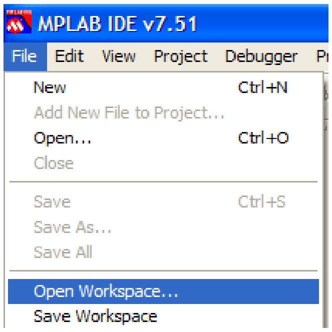
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab16\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab16\...

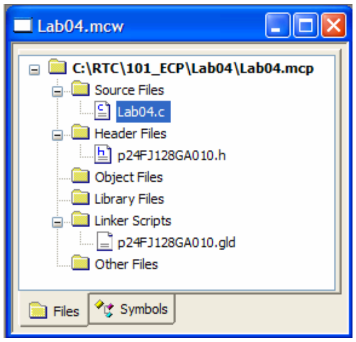
Procedure

On the class PC:
C:\RTC\101_ECP\Lab16\Lab16.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

⚠ If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.



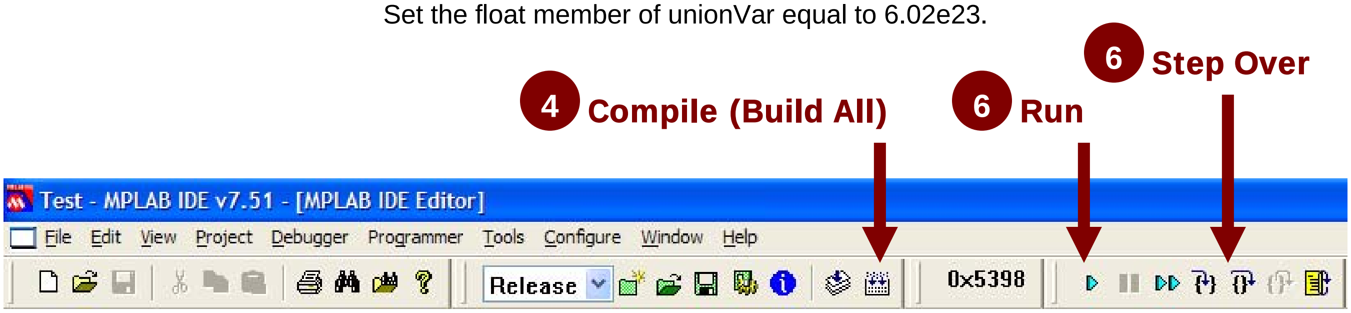
2 If it isn't already visible in the workspace, open the Lab16.c source file from the project tree by double clicking on its icon.

Alternatively, you may right click the icon and select "Edit" from the popup menu.

3 **Edit source code as instructed in the comments**

STEP 1:
Set the int member of unionVar equal to 16877.

STEP 2:
Set the float member of unionVar equal to 6.02e23.

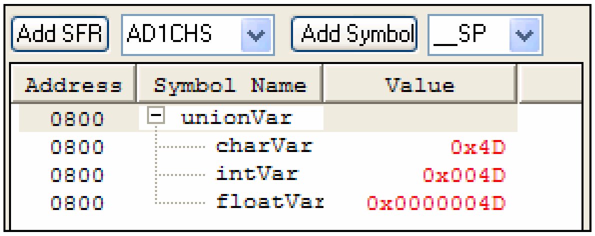

6 Step Over
4 Compile (Build All)
6 Run
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

4 Click on the **Build All** button.

5 Set breakpoints on line 63 and the two lines where you added your own code by double clicking on the line (alternatively, right click on the line and select "**Breakpoints** ▶ **Set Breakpoint**"). You should then see a small red octagon with a 'B' to the left of the line: B

6 Click on the **Run** button, followed by the **Step Over** button.

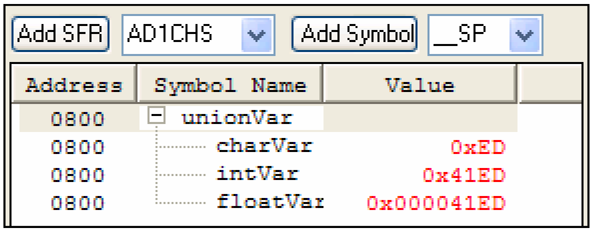
You should now see the following values at right in the watch window.
Note that even though we only wrote to the character member of the union, all three members changed and have the same value. (0x4D is the ASCII value of 'M')



Address	Symbol Name	Value
0800	⊟ unionVar	
0800	charVar	0x4D
0800	intVar	0x004D
0800	floatVar	0x0000004D

7 Click on the **Step Over** button.

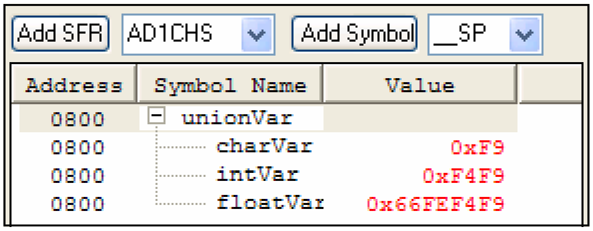
Now, once again all three members changed to reflect the value that we wrote to the integer member of the union. (0x41ED = 16877_{10})



Address	Symbol Name	Value
0800	⊟ unionVar	
0800	charVar	0xED
0800	intVar	0x41ED
0800	floatVar	0x000041ED

8 Click on the **Step Over** button.

Finally, we write a value to the floating point member of the union, and as you probably expected by now, all three members values changed. As you can see, the three members do share the same memory. All three share the lowest byte and the int and float share the next lowest byte.
(0x66FEF4F9 is the IEEE floating point formatted version of 6.02e23)



Address	Symbol Name	Value
0800	⊟ unionVar	
0800	charVar	0xF9
0800	intVar	0xF4F9
0800	floatVar	0x66FEF4F9

Results

If your code was correct, then you should have seen the values shown in each of the steps above.

Code Analysis

Lines 31-35:
This is the union variable declaration. There are three members of three types: char, int and float. All three of these members will share memory locations. This is most obviously seen in step 8 above, where the lowest byte of all three members share the same value and the second lowest byte of the int and float members share the same value.

Line 63:
This line sets the char member to a value of 'M' (ASCII 0x4D)

Line 69:
STEP 1: This line should look like line 63, but here we are writing to the intVar member.

```
unionVar.intVar = 16877;
```

Line 75:
STEP 2: Similar to line 69, but this time we are writing to the floatVar member.

```
unionVar.floatVar = 6.02e23;
```

Lab 17

Bit Fields—Demo

Purpose

This demo will illustrate the use of bit fields. There is no code for you to write. All you need to do is build and run the project and observe the results.

The code itself combines what we learned in Lab 16 about unions, with the concept of bit fields to create a variable that will allow us to access it as a full byte, or as individual bits.

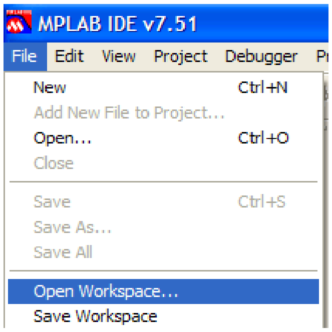
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab17\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab17\...

Procedure

On the class PC:
C:\RTC\101_ECP\Lab17\Lab17.mcw



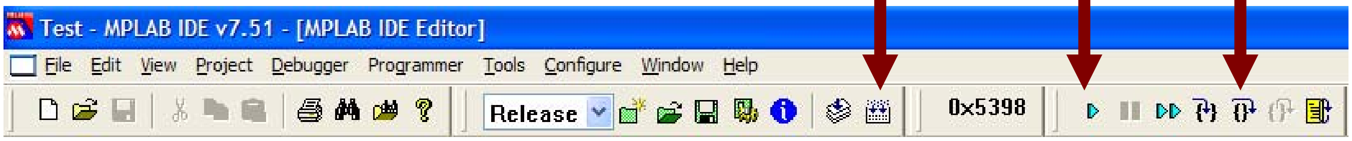
1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.

4 **Step Over**

2 **Compile (Build All)**

4 **Run**


Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

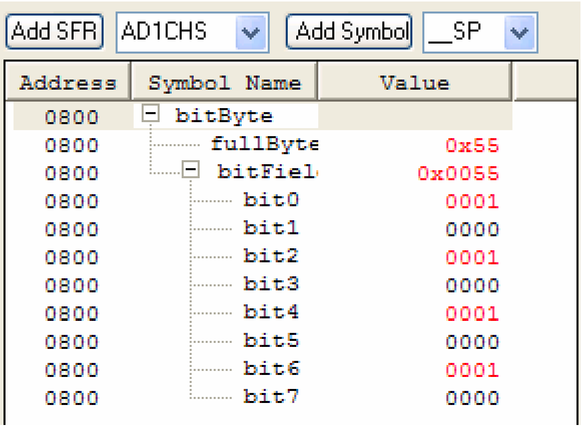
2 Click on the **Build All** button.

3 Set a breakpoint on line 68 by double clicking on the line. (Alternatively, right click on the line and select "**Breakpoints** ▶ **Set Breakpoint**"). You should then see a small red octagon with a 'B' to the left of the line: B

4 If no errors are reported, click on the **Run** button, followed by the **Step Over** button.

Line 68 has just been executed, so we wrote a value of 0x55 to the fullByte member of the variable bitByte. Note that the bitField members also changed appropriately to reflect the new value of 0x55 = 0b01010101.

```
bitByte.fullByte = 0x55;
```

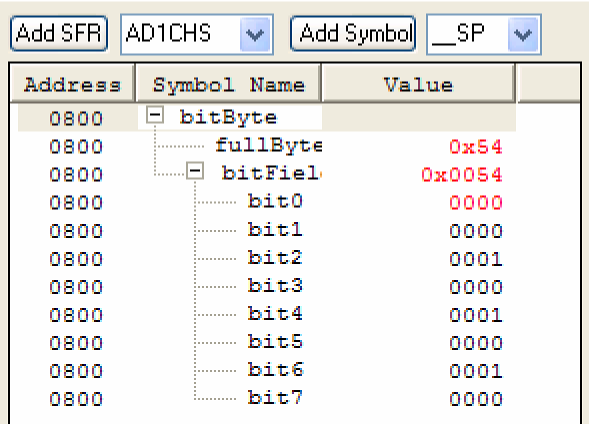


Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0x55
0800	⊟ bitFiel	0x0055
0800	bit0	0001
0800	bit1	0000
0800	bit2	0001
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0000

5 Click on the **Step Over** button.

Line 69 has just been executed, where we wrote a value of 0 to the member bit0 of bitField, which itself is a member of the union bitByte. Therefore, when we changed the individual bit, the fullByte member also changed to reflect the new value.

```
bitByte.bitField.bit0 = 0;
```

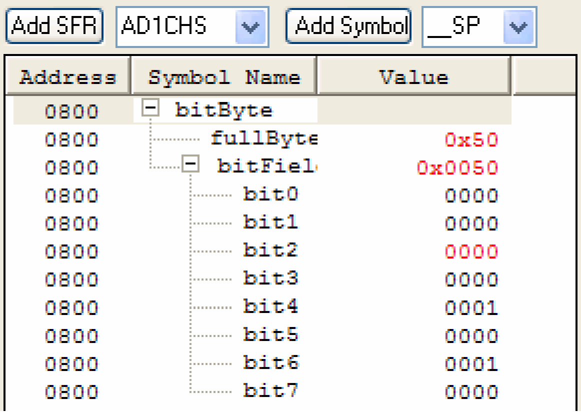


Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0x54
0800	⊟ bitFiel	0x0054
0800	bit0	0000
0800	bit1	0000
0800	bit2	0001
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0000

6 Click on the **Step Over** button.

Line 70 has just been executed, where we wrote a value of 0 to the member bit2 of bitField, which itself is a member of the union bitByte. Therefore, when we changed the individual bit, the fullByte member also changed to reflect the new value.

```
bitByte.bitField.bit2 = 0;
```

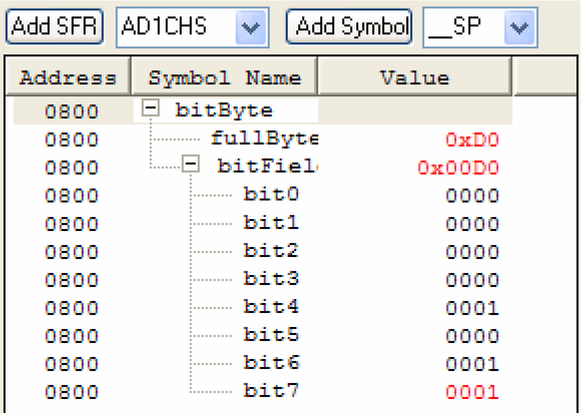


Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0x50
0800	⊟ bitFiel	0x0050
0800	bit0	0000
0800	bit1	0000
0800	bit2	0000
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0000

7 Click on the **Step Over** button.

Line 70 has just been executed, where we wrote a value of 1 to the member bit7 of bitField, which itself is a member of the union bitByte. Therefore, when we changed the individual bit, the fullByte member also changed to reflect the new value.

```
bitByte.bitField.bit7 = 1;
```



Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0xD0
0800	⊟ bitFiel	0x00D0
0800	bit0	0000
0800	bit1	0000
0800	bit2	0000
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0001

Conclusions

Bit fields allow us to efficiently use individual bits for Boolean values or as flags/semaphores. On the various PIC architectures, setting and clearing a bit field variable in C will make use of the very efficient bit set and bit clear instructions in assembly language. However, other operations may generate more code than would be necessary if you were working with a full 16-bit integer type variable. So, bit fields can be invaluable in some circumstances, but they should be used with care so that excess code will not be generated.

Lab 18

Enumerations—Demo

Purpose

This demo will illustrate the use enumerations to create a list of constant labels that may be used in conjunction with variables declared with the enum’s type. The primary purpose of enumerations is to make your code more readable and easier to maintain.

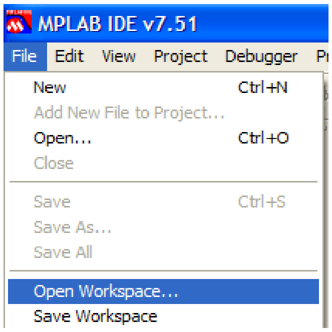
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab18\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab18\...

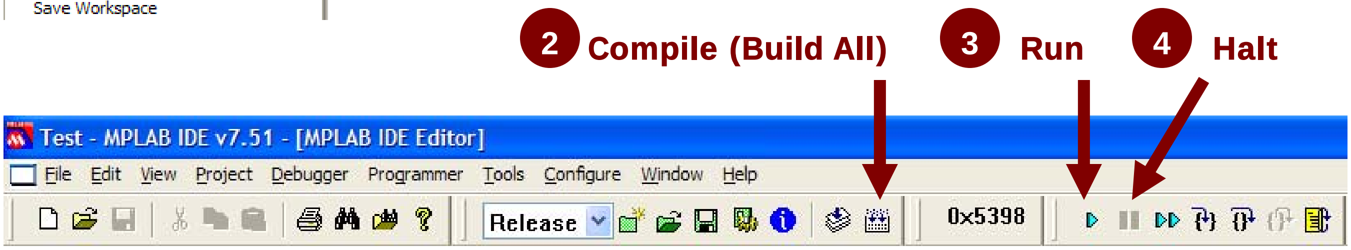
Procedure

On the class PC:
C:\RTC\101_ECP\Lab18\Lab18.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.


2 Compile (Build All)
3 Run
4 Halt
Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button.

This will run the program in the simulator at full speed.

4 Click on the **Halt** button.

This will stop execution so that we may examine the variables and their values.

Results

After running the program, the Sim Uart1 window should display the text: “Bandpass filter selected.”

Code Analysis

Line 28:
An enumeration data type called `filterTypes` is defined. This line essentially defines the following constants:
BANDSTOP = 0
LOWPASS = 1
HIGHPASS = 2
BANDPASS = 3
While these constants may be used just about anywhere, they are intended to be used with variables that are

declared to be of type `filterTypes`.

Line 33:
This line declares a variable called `filter` of type `filterTypes`. Many compilers will restrict this variable to holding just the values defined by the enumeration, though it isn't always the case. At the very least, it will make your code easier to read and maintain by associating a group of constants with a particular variable.

Line 49:
Here, we initialized the variable filter to BANDPASS. This effectively assigns a value of 3 to the variable, which may be used as any ordinary integer type variable.

Lines 51-57:
These lines simply call a particular function depending on the value of the variable `filter`. Notice that in every place that a constant would be used, we have used on of the labels defined in the enumeration type declaration. The functions themselves are defined in the file Utilities.c, and all they do is print out which filter was selected to the Sim Uart1 window.

Conclusions

Enumerations provide a means to associate a list of constants with one or more variables. Theoretically, these constants represent the entire range of valid values for the variable. Some compilers will enforce this range (compile time only—runtime checking is up to you). However, even if the compiler doesn't do any checking, by sticking to the list of valid constants it becomes more difficult to accidentally assign an invalid value.

Perhaps the greatest benefit of using enumerations is that they make your code more readable by replacing "magic numbers" and that it becomes much easier to maintain. If you need to add additional valid values, you need only add them to the enum list. If the value of some of the labels change, you may not have to change the rest of your code to reflect the changes (depending on how you have written your code).

Lab 19

Macros—Demo

Purpose

This demo will illustrate some of the many uses for macros. Macros created with the #define directive can help simplify and add flexibility to your code. Macros are operations that will be performed by the compiler when it is building your code. So anything you could compute at compile time can be handled by the compiler for you.

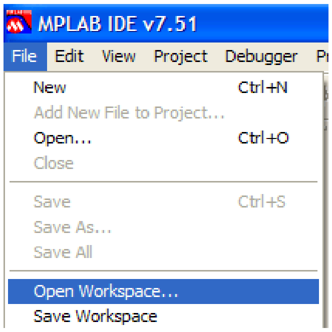
Requirements

Development Environment:	MPLAB 7.51 or later
C Compiler:	MPLAB C30 2.04 or later (Free student version works too)
Lab files on class PC:	C:\RTC\101_ECP\Lab19\...

Lab files, including solutions are included on the CD:
…\101_ECP\Lab19\...

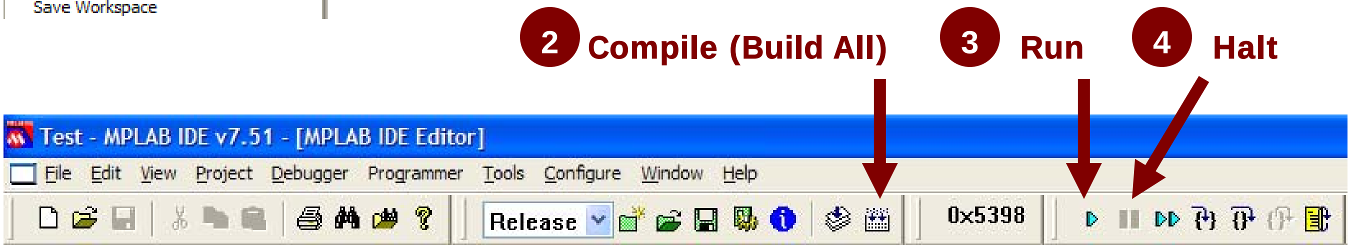
Procedure

On the class PC:
C:\RTC\101_ECP\Lab19\Lab19.mcw



1 Open MPLAB® and select **Open Workspace...** from the **File** menu. Open the file listed above.

If you already have a project open in MPLAB®, close it by selecting **Close Workspace** from the **File** menu before opening a new one.


2 **Compile (Build All)** 3 **Run** 4 **Halt**

Test - MPLAB IDE v7.51 - [MPLAB IDE Editor]
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
0x5398

2 Click on the **Build All** button.

3 If no errors are reported, click on the **Run** button. This will run the program in the simulator at full speed.

4 Click on the **Halt** button. This will stop execution so that we may examine the variables and their values.

Results

After running the program, the Sim Uart1 window should display the text:
x = 9
SPBRG = 25

Code Analysis

Line 27:
This first macro may be used to compute the square of any variable passed to it. Some care must be exercised when using it (as discussed in the presentation), but using macros like this can simplify your code and make it more readable:

```
#define square(m) ((m) * (m))
```

It may be used in your code much like a function. The benefit of using a macro like this is that it doesn't care what data type the parameter has. The disadvantage is that the code will be compiled inline and take up more space than a function call, if it is used frequently.

Line 28:
This handy utility macro may be used to calculate the value to write to a UART's baud rate generator's control register. This particular macro is intended for the PIC18 family, but could be modified easily for other PIC families.

```
#define BaudRate(DesiredBR, FoscMHz) ((((FoscMHz * 1000000)/DesiredBR)/64)-1)
```

Note that the desired baud rate (DesiredBR) and oscillator frequency (FoscMHz) must be known at compile time. In other words, you can only pass constants to a macro. Trying to pass a variable, whose value cannot be known at compile time, will produce an error.
Using a macro like this makes it much easier to change the parameters at design time, or to make your code more easily customizable for the future.

Line 49:
The square() macro is invoked here by passing the value 3 to it. It will return a value of 9 to be stored in the variable x.
x = square(3);
This line of code will not generate any extra overhead. It will be the same as if you had written:
x = 9;

Line 52:
This line is used to calculate the value to be loaded into the SPBRG register, which controls the baud rate on a of the USART on a PIC18:

```
SPBRG = BaudRate(9600, 16);
```

Just like above, no extra overhead is generated here. This line will generate the same assembly code as:

```
SPBRG = 25;
```

Conclusions

Much like enumerations, macros can make your code more readable and easier to maintain. However, unlike enumerations, there is much more potential for misuse. So, extreme care must be exercised when writing a macro and when invoking a macro.