**CAMOSUN COLLEGE**

**ELECTRONICS DEPTARTMENT**

**ECET 165 - LAB 5**

**XC8 C Compiler Introduction**

**REF: MPLAB**® **XC8 C Compiler User’s Guide**

The follow is condensed version of the MPLAB® XC8 C Compiler User’s Guide. It should get you started with most of the C labs. Please refer to original as needed.

<https://ww1.microchip.com/downloads/en/DeviceDoc/50002737C%20XC8%20C%20Compiler%20UG%20for%20PIC.pdf>

<https://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB%20XC8%20C%20Compiler%20UG%20EE%20DS50002400C%20.pdf>

**Part 1 Read the following and consider how to apply it to part 2:**

The following shows an example of how main() might be defined:

#include <xc.h>

int main(void)  
{

\\ your setup code  
 while(1)

{  
 // your main loop of code

}  
}

Each program has one definition for the main() function.

**Plain char Types**

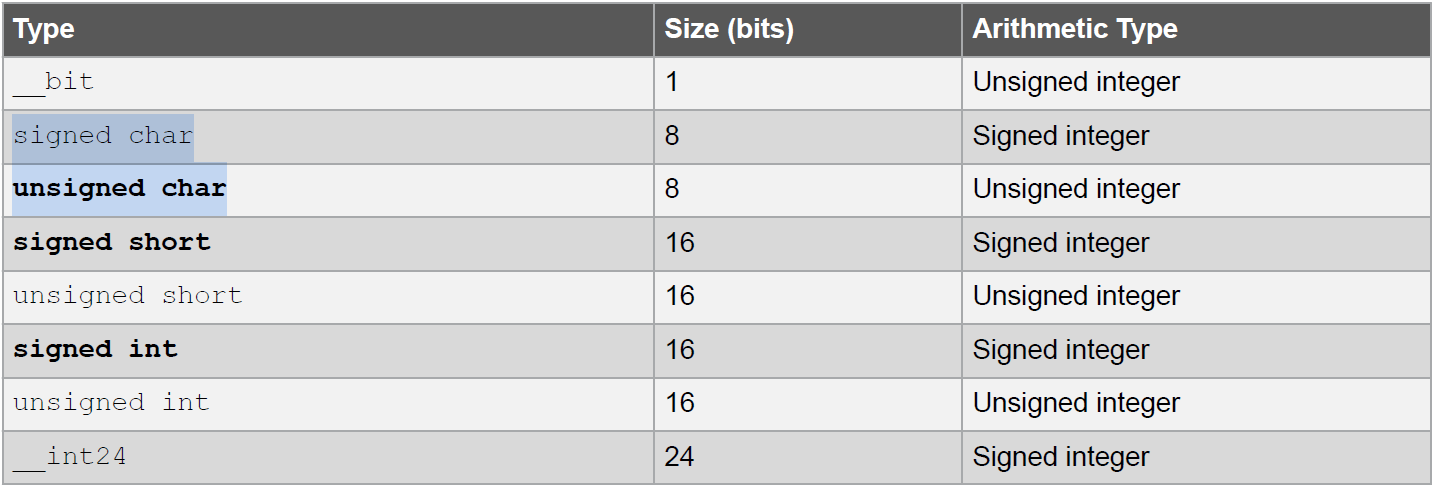
The type of a plain char is unsigned char. It is generally recommended that all definitions for the char type explicitly state the signed-ness of the object.

Always use an unsigned char unless you have a good reason!!!

**5.3.2 Integer Data Types**

The MPLAB XC8 compiler supports integer data types with 1, 2, 3, 4 and 8 byte widths as well as a single bit type.

Table 5-3. Integer Data Types



**5.2.3 Device Header Files**

There is one header file that is typically included into each C source file you write. The <xc.h> file is a generic header file that will include other device- and architecture-specific header files when you build your project.

#include <xc.h>

**5.2.5 Configuration Bit Access**

Configuration bits, or fuses, are used to set up fundamental device operation, such as the oscillator mode, watchdog timer, programming mode and code protection. These bits must be correctly set to ensure your program executes correctly.

Use the configuration pragma, which has the following forms, to set up your device.

#pragma config FEXTOSC = OFF // External Oscillator Selection (Oscillator not enabled)

#pragma config RSTOSC = HFINTOSC\_64MHZ // Reset Oscillator Selection (HFINTOSC with HFFRQ = 64 MHz and CDIV = 1:1)

#pragma config MCLRE = EXTMCLR // MCLR Enable bit (If LVP = 0, MCLR pin is MCLR; If LVP = 1, RE3 pin function is MCLR )

#pragma config BOREN = OFF // Brown-out Reset Enable bits (Brown-out Reset disabled)

#pragma config LVP = ON // Low Voltage Programming Enable bit (Low voltage programming enabled. MCLR/VPP pin function is MCLR.

//MCLRE configuration bit is ignored)

#pragma config WDTE = OFF // WDT operating mode (WDT Disabled; SWDTEN is ignored)

**5.11.2 Inline Assembly**

Assembly instructions can be directly embedded in-line into C code using the statement asm();. The instructions are placed in a string inside what look like function call brackets, although no actual call takes place. Typically one instruction is placed in the string, but you can specify more than one assembly instruction by separating the instructions with a \n character,

e.g., asm(“movlw 55\nmovwf \_x”);,

code will be more readable if you place one instruction in each statement and use multiple statements. You can use the asm() form of in-line assembly at any point in the C source code as it will correctly interact with all C flow-of-control structures, as shown below.

unsigned int var;

int main(void)

{

var = 1;

asm(“bcf 0,3”);

asm(“BANKSEL \_var”);

asm(“rlf (\_var)&07fh”);

asm(“rlf (\_var+1)&07fh”);

}

**3.3 Writing Source Code**

**3.3.1.4 How Can I Use A Variable Defined In Another Source File?**

Provided the variable defined in the other source file is not specified static or auto, then adding a declaration (as opposed to a definition) for that variable into the current file will allow you to access it. A declaration consists of the keyword extern in addition to the correct type and the exact name of the variable specified in its definition, e.g.,

extern int systemStatus; // declare systemStatus for use here

**5.2.7 Using SFRs From C Code**

The Special Function Registers (SFRs) are typically memory mapped registers and are accessed by absolute C structure variables that are placed at the register’s address. These structures can be accessed in the usual way so that no special syntax is required to access SFRs. The SFRs control aspects of the MCU and peripheral module operation. Some registers are read-only; some are write-only. Always check your device data sheet for complete information regarding the registers.

The SFR structures are predefined in header files and are accessible once you have included <xc.h> (see 5.2.3 Device Header Files) into your source files. Structures are mapped over the entire register and bit-fields within those structures allow access to specific SFR bits. The names of the structures will typically be the same as the

corresponding register, as specified in the device data sheet, followed by bits (see 3.3.2.5 How Do I Find The Names Used To Represent SFRs And Bits?). For example, the following shows code that includes the generic header file, clears PORTA as a whole and sets bit 2 of PORTA using the bit-field definition.

#include <xc.h>

int main(void)

{

PORTA = 0x00;

PORTAbits.RA2 = 1;

}

**3.4.1 What Can Cause Glitches on Output Ports?**

In most cases, this is caused by using ordinary variables to access port bits or the entire port itself. These variables should be qualified **volatile**. The value stored in a variable mapped over a port (hence the actual value written to the port) directly translates to an

electrical signal. It is vital that the values held by these variables only change when the code intends them to and that they change from their current state to their new value in a single transition (see 5.3.8.2 Volatile Type Qualifier). The compiler attempts to write to **volatile** variables in one operation.

**3.6.1 How Do I Find Out What an Warning/Error Message Means?**

Each warning or error message has a description and possibly sample code that might trigger such an error, listed in the messages chapter (see 10. Error and Warning Messages).

**3.6.2 How Do I Find the Code that Caused Compiler Errors or Warnings in My Program?**

In most instances, when a syntax error occurs that relates to the source code, the message produced by the compiler indicates the offending line of code (see 4.5.1 Messaging Overview). If you are compiling in MPLAB X IDE, then you can double-click the message and have the editor take you to the offending line. But identifying the offending code is not always so easy. In some instances, the error is reported on the line of code following the line that needs attention. This is because a C statement is allowed to extend over multiple lines of the source file. It is possible that the compiler cannot be able to determine that there is an error until it has started to scan to statement following. So in the following code input = PORTB // oops - forgot the semicolon

if(input>6)

// ...

The missing semicolon on the assignment statement will be flagged on the following line that contains the if() statement. In other cases, the error might come from the assembler, not the code generator. If the assembly code was derived from a C source file, then the compiler will try to indicate the line in the C source file that corresponds to the assembly that is at fault. If the source being compiled is an assembly module, the error directly indicates the line of assembly that triggered the error. In either case, remember that the information in the error relates to some problem is the assembly code, not the C code.

Finally, there are errors that do not relate to any particular line of code at all. An error in a compiler option or a linker error are examples of these. If the program defines too many variables, there is no one particular line of code that is at fault; the program as a whole uses too much data. Note that the name and line number of the last processed file and source can be printed in some situations even though that code is not the direct source of the error.

To determine the application that generated the error or warning, check the message section of the manual, see 10. Error and Warning Messages. At the top of each message description, on the right in brackets, is the name of the application that produced this message. Knowing the application that produced the error makes it easier to track down the problem. The compiler application names are indicated in 4.2 The Compilation Sequence. If you need to see the assembly code generated by the compiler, look in the assembly list file (see 6.3 Assembly List Files). For information on where the linker attempted to position objects, see the map file discussed in 7.3 Map Files.

**3.6.4 Why Can’t I Even Blink an LED?**

Even if you have set up the TRIS register and written a value to the port, there are several things that can prevent such a seemingly simple program from working.

• Make sure that the device’s Configuration registers are set up correctly (see 5.2.5 Configuration Bit Access).

Make sure that you explicitly specify every bit in these registers and don’t just leave them in their default state.

All the configuration features are described in your device data sheet. If the Configuration bits that specify the oscillator source are wrong, for example, the device clock cannot even be running.

• If the internal oscillator is being used, in addition to Configuration bits there can be SFRs you need to initialize to set the oscillator frequency and modes, see 5.2.7 Using SFRs From C Code and your device data sheet.

• Either turn off the Watchdog Timer in the Configuration bits or clear the Watch Dog Timer in your code so that the device does not reset. If the device is resetting, it can never reach the lines of code in your program that blink the LED. Turn off any other features that can cause device Reset until your test program is working.

• The device pins used by the port bits are often multiplexed with other peripherals. A pin might be connected to a bit in a port, or it might be an analog input, or it might the output of a comparator, for example. If the pin connected to your LED is not internally connected to the port you are using, then your LED will never operate as expected. The port function tables shown in your device data sheets will show other uses for each pin that will help you identify peripherals to investigate.

• Make sure you do not have a “read-modify-write” problem. If the device you are using does not have a separate “latch” register (as is the case with mid-range PIC devices) this problem can occur, particularly if the port outputs are driving large loads, such as an LED. You can see that setting one bit turns off another or other unusual events. Create your own latch by using a temporary variable. Rather than read and write the port directly, make modifications to the latch variable. After modifications are complete, copy the latch as a whole to the port. This means you are never reading the port to modify it. Check the device literature for more detailed information.

**Part 2:**

**Rotate LED**

In this document we will review some of the structures that we will need to complete this exercise. Firstly good documentation is essential. All files need to have a declaration at the start of the file.

/\*

Title: lab5rotateLED.c

Author: Your Name

Date: July 4, 2031

Description: Causes one LED to be lit and then rotated through all bits of Port C. On reaching the end, it then will rotate back in the opposite direction. Program will repeat forever.

Compiled with MPLAB XC8 compiler ver X.xx

and MPLAB X ver X.xx

\*/

#include <xc.h> /\* for TRISC and PORTC declarations \*/

main(void)

{

/\* Your code \*/

}

The above title block, includes basic information. Some people like to put banners around it so that it stands out in the code but this is optional.

As shown above, it’s pretty consistent with other programs.

So getting back to the lab, there are three things that we need to accomplish in this program.

1. Create a setup and an infinite loop
2. Rotate an LED through all port pins of PORTC
3. Create a time delay that will allow us to see the LED move

**Step 1 Create an infinite loop:**

The first one is common to all embedded programs. On power up we need the code to execute until the device is powered down. When writing console type C programs on a PC we always leave a way to break out of a loop, a variable reaches a required value or a key press is detected. If not you would have to terminate the program with a Ctrl+Break.

In an embedded program it is the opposite situation, we want the main program to be continually executed.

We can make use of the While command for the infinite loop

while (expression)

{

...block of statements to execute...

}

The expression part just has to be a logic true to continue execution of the contents of the braces which lead us to the following.

while (1)

{

...block of statements to execute...

}

**Step 2 Rotate an LED through all port pins of PORTC:**.

We will need to use two for loops to help with the rotating of the lit LED…

for (expression\_1; expression\_2; expression\_3)

{

...block of statements to execute...

}

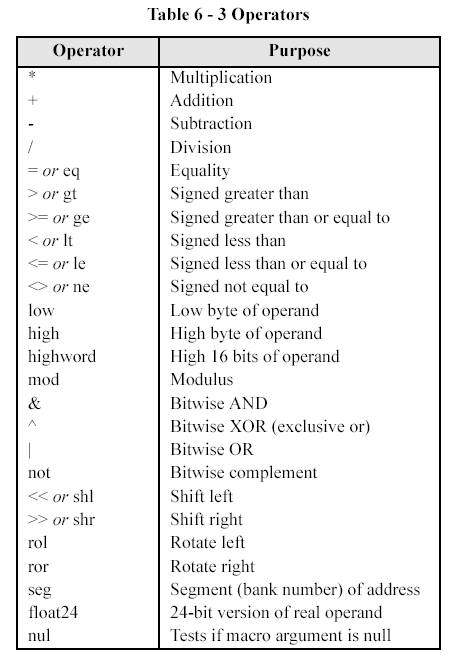
expression\_1 is the initial condition for a variable, expression\_2 is the terminating condition and expression\_3 is the increment direction and magnitude.

What we have to try and do here, should make you think of shift registers. If you load lets say a H in the D0, and clock the shift register, the H will ripple through each output in turn. When it gets to the end however, we need to reverse the direction. In assembly language there are two commands that will rotate the contents of a register in either direction. RLNCF and RRNCF.

We can use them from C by using the following line of Code which is interpreted as assembly statement by the complier. The asm() statement can be used to insert assembly code in-line with C code:

asm("RLNCF PORTC,F");

However be warned, when using inline assembler code, great care must be taken to avoid interacting with compiler generated code. If in doubt, compile your program with the generate assembly option and examine the assembler code generated by the compiler.



The above table lists some of the allowable operators. Note that there are a couple of easy options that will help in rotating our LED’s

If we look at the Shift left operator it functions in the following way.

LATC = 0b00000001<< N

The above would result in a 1 which would be initially in Bit0 rotated left N times and then the result written out to PORTC. This operator is probably the best approach to this problem.

**Step 3 Use a time delay that will allow us to see the LED move:**

If we write the program without including a delay, the LED’s will all be lit but will be dimmer than they could be. This is because our code will be executing faster than our eyes can see the LED’s. We need to slow it down. The lab spec does not call for a specific speed (yet!) so we can use a simple delay tactic.

**Optional:**

Write other C programs that use switches and LEDs. Demo the most advanced program you create.